Memoria Técnica - Sistema de Ingesta y Consulta de Datos

Alejandro Ruiz

Índice

1. Objetivo General del Sistema

El objetivo de este sistema distribuido es proporcionar a clientes externos acceso a datos meteorológicos procesados y actualizados, a través de una arquitectura de microservicios. Para ello se diseñaron dos servicios que cooperan: un **Servicio A** encargado de la *ingesta* y almacenamiento de datos desde un fichero CSV en una base de datos relacional, y un **Servicio B** que ofrece una *API REST* para consultas agregadas de esos datos (con cálculos estadísticos y conversión de unidades). Ambos servicios se comunican internamente mediante llamadas HTTP con datos en formato JSON, y el sistema completo está empaquetado mediante contenedores Docker para facilitar su despliegue.

2. Arquitectura del Sistema

La solución adopta una arquitectura de microservicios, con componentes desacoplados que interactúan sobre la red local Docker. En la Figura 1 se ilustra la estructura general: los usuarios realizan peticiones al Servicio B (API de consulta), el cual a su vez consulta al Servicio A (almacenamiento) y utiliza un caché para optimizar el rendimiento. El Servicio A persiste los datos en una base de datos PostgreSQL, mientras que el Servicio B utiliza Redis como caché de respuestas recientes para reducir accesos repetitivos a la base de datos. Todo el conjunto (Servicio A, Servicio B, PostgreSQL y Redis) es orquestado por *Docker Compose*, incluyendo comprobaciones automáticas de salud (*healthchecks*) para garantizar el inicio ordenado (por ejemplo, el Servicio A espera a que la base de datos esté lista antes de arrancar).

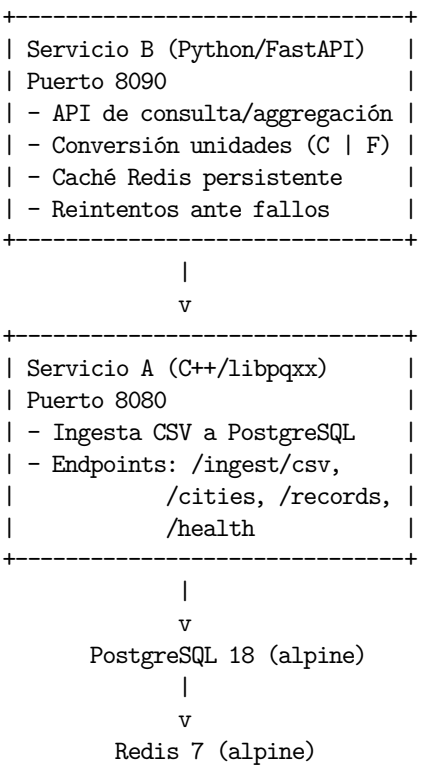

Figura 1: Arquitectura general del sistema de ingestión y consulta de datos.

Comunicación interna: El Servicio B actúa como fachada del sistema frente a los clientes. Cuando recibe una solicitud de datos meteorológicos, contacta al Servicio A a través de una petición HTTP interna (por ejemplo, usando `http://servicioa:8080` dentro de la red Docker) para obtener los registros crudos necesarios. De esta manera, el Servicio A se encarga de leer

desde la base de datos los datos solicitados, y el Servicio B de procesarlos y presentarlos en el formato requerido por el cliente. Ambos servicios manejan JSON tanto en las peticiones como en las respuestas para mantener la interoperabilidad y la claridad en el intercambio de información.

3. Servicio A: Ingesta y Almacenamiento

El Servicio A es una API REST desarrollada en C++ (usando la biblioteca `httplib` para el servidor HTTP y `libpqxx` para la conexión con PostgreSQL). Su responsabilidad principal es recibir archivos CSV con datos meteorológicos, normalizarlos y almacenarlos en la base de datos, así como proveer acceso de lectura en crudo a esos datos. A continuación se resumen sus endpoints principales en el Cuadro 1:

Método	**Endpoint**	**Descripción**
GET	`/health`	Comprobación de estado (BD conectada)
POST	`/ingest/csv`	Ingesta un fichero CSV en la base de datos
GET	`/cities`	Lista las ciudades disponibles
GET	`/records`	Consulta registros crudos (por ciudad y rango)

Cuadro 1: Endpoints principales del Servicio A.

Ingesta de CSV (`POST /ingest/csv`): Este endpoint recibe un fichero CSV con datos meteorológicos (ya sea mediante *multipart/form-data* con un campo de archivo, o directamente en el cuerpo de la petición con formato texto). El servicio realiza varias tareas de normalización y validación:

- Convierte la fecha de cada registro al formato ISO `YYYY-MM-DD` y verifica que esté en un rango válido.
- Normaliza los valores numéricos (temperaturas máximas/mínimas en grados Celsius, precipitación en milímetros, nubosidad en %); por ejemplo, se aceptan separadores decimales con coma o punto y se eliminan espacios innecesarios.
- Valida cada fila descartando datos inconsistentes: ciudad vacía, valores fuera de rango (e.g. porcentaje de nubosidad mayor que 100 %, precipitaciones negativas), o temperaturas mínima y máxima incoherentes (la mínima no puede exceder a la máxima).

Las filas válidas se almacenan en la tabla `weather_readings` de PostgreSQL. Dicha tabla contiene las columnas: `date` (fecha), `city` (ciudad), `temp_max`, `temp_min`, `precip_mm` (precipitación) y `cloud_pct` (nubosidad), con una restricción de unicidad compuesta en `(city, date)` para evitar duplicados. La inserción se realiza fila a fila utilizando sentencias `INSERT ... ON CONFLICT DO NOTHING`, de modo que si el CSV contiene registros ya existentes (misma ciudad y fecha) no se duplicarán en la base de datos.

Al finalizar la operación de ingesta, el Servicio A responde con un resumen en JSON que incluye:

- `rows_inserted`: número de filas insertadas exitosamente en la base de datos (nuevos registros).
- `rows_rejected`: número de filas rechazadas (por formato inválido o datos fuera de rango) *más* aquellas no insertadas por estar duplicadas (conflicto de unicidad).
- `elapsed_ms`: tiempo total empleado en la ingesta (en milisegundos).
- `file_checksum`: un hash SHA-256 del fichero CSV ingerido (útil para identificar el archivo cargado).

Consulta de ciudades (`GET /cities`): Devuelve la lista de todas las ciudades distintas cuyos datos se encuentran almacenados. Esta operación realiza una consulta SQL de tipo `SELECT DISTINCT` sobre la columna `city` de la tabla, ordenando alfabéticamente los nombres. En caso de que la base de datos no esté accesible, responde con un error 503.
Consulta de registros crudos (`GET /records`): Permite obtener los datos meteorológicos en crudo de una ciudad, dentro de un rango de fechas específico, con soporte de paginación. Los parámetros de consulta son: *city* (nombre de la ciudad, obligatorio), *from* (fecha de inicio, formato `YYYY-MM-DD`, obligatorio), *to* (fecha de fin, obligatorio), *page* (número de página, opcional, por defecto 1) y *limit* (cantidad de resultados por página, opcional, por defecto 10, máximo 100). El servicio valida que las fechas tengan el formato correcto y `from` no sea posterior a `to`. Luego realiza dos consultas a la base de datos: una para contar el total de registros que cumplen los criterios (para cálculo de *paginación*), y otra para obtener la página solicitada de resultados. Los datos devueltos incluyen meta-información de paginación: `page`, `limit`, `total` (total de registros hallados) y `total_pages`, junto con un arreglo `items` que contiene los registros (cada uno con su fecha, temperatura máxima, temperatura mínima, precipitación y nubosidad). Los resultados vienen ordenados por fecha ascendente. Un ejemplo de respuesta sería:

```
{
  "city": "Madrid",
  "from": "2025-10-01",
  "to": "2025-10-31",
  "page": 1,
  "limit": 10,
  "total": 20,
  "total_pages": 2,
  "items": [
    {"date": "2025-10-12", "temp_max": 11.55, "temp_min": 6.25, "precip_mm": 0.0, "cloud_pct": 10},
    {"date": "2025-10-13", "temp_max": 12.35, "temp_min": 5.25, "precip_mm": 0.2, "cloud_pct": 60},
    ...
  ]
}
```

El Servicio A también expone un endpoint `/health` (no listado en la respuesta anterior) que simplemente verifica la conexión a la base de datos y devuelve un estado `200 OK` con `{"status":"DB OK"}` si la conexión está activa (o un error 503 si hay problemas de conexión). Este endpoint es utilizado por Docker Compose para el *healthcheck* automático del contenedor.

4. Servicio B: API de Consulta Agregada

El Servicio B está desarrollado en Python utilizando FastAPI (Uvicorn como servidor ASGI) y ofrece una API REST en el puerto 8090 para consultar los datos meteorológicos de forma agregada. Funciona como orquestador de las consultas: recibe peticiones de clientes externos, recupera los datos básicos a través del Servicio A y luego aplica agregaciones, conversión de unidades y caching. Los endpoints principales se resumen en el Cuadro 2.

Método	Endpoint	Descripción
GET	`/health`	Estado del servicio (y tipo de caché en uso)
GET	`/weather/{city}`	Consulta meteorológica (cruda o agregada)

Cuadro 2: Endpoints principales del Servicio B.

Endpoint principal (`GET /weather/{city}`): Este endpoint proporciona información meteorológica de la ciudad indicada, permitiendo diferentes modalidades de consulta mediante parámetros de *query*:

- `date`: fecha inicial de los datos requeridos (obligatorio, formato `YYYY-MM-DD`).
- `days`: número de días a consultar a partir de la fecha inicial (entero, por defecto 5, permitido 1 hasta 10 días).
- `unit`: unidad de temperatura deseada, `C` para Celsius o `F` para Fahrenheit (opcional, defecto `C`).
- `agg`: tipo de agregación a aplicar, `daily` (agregación diaria) o `rolling7` (media móvil de 7 días). Este parámetro es opcional; si no se proporciona, el resultado será una lista de datos diarios crudos (sin agregación) para el rango especificado.

Al procesar una petición `/weather/{city}`, el Servicio B sigue estos pasos de forma interna:

1. **Validación inicial:** Verifica el formato de la fecha inicial dada y calcula la fecha final sumando `days`-1 días. Por ejemplo, si `date=2025-10-15` y `days=5`, el rango cubierto será del 15 al 19 de octubre de 2025 inclusive.
2. **Consulta a caché:** Se construye una clave única para esta consulta (por ejemplo, `Madrid:2025-10-15:5:C:daily`) y se busca en la caché Redis si ya existe una respuesta guardada para esos parámetros. Si la caché contiene un resultado válido ("`HIT`"), se devuelve inmediatamente esa respuesta almacenada, evitando cálculos repetidos.
3. **Obtención de datos crudos (Servicio A):** Si no hay resultado en caché ("`MISS`"), el Servicio B realiza una petición HTTP interna al Servicio A (`/records`) para obtener los registros diarios crudos de la ciudad y rango especificados. Esta llamada se hace de forma asíncrona usando un cliente HTTP (biblioteca `httpx`), con un tiempo de espera configurado (timeout) y una política de reintentos en caso de fallo:
   - Si el Servicio A responde con éxito (`200 OK`), se procede con los datos recibidos.
   - Si la respuesta es un error de cliente (`4xx`, por ejemplo ciudad no encontrada o parámetros inválidos), no se reintenta y simplemente se propaga ese error al cliente final.
   - Si la respuesta indica un error de servidor o no hay respuesta (conexión rechazada, tiempo de espera excedido, código `5xx`), el Servicio B realizará reintentos automáticos con *backoff* exponencial y *jitter* (esperas crecientes aleatorizadas) antes de declarar la falla. Por configuración, se intentan hasta 3 reintentos. En cada intento, el retraso aumenta (1s, 2s, 4s, etc.). Si tras agotar los intentos el Servicio A sigue inaccesible, el Servicio B verifica si *en caché* hay una respuesta previa para esa misma consulta; de ser así, la sirve al cliente (asegurando cierta continuidad del servicio con datos potencialmente desactualizados). Si tampoco hay datos caché disponibles, entonces responde al cliente con un error 503 indicando que el Servicio A no está disponible.
4. **Procesamiento de agregación:** Con los datos crudos obtenidos (lista de registros diarios con campos `date`, `temp_max`, `temp_min`, `precip_mm`, `cloud_pct`), el Servicio B aplica la agregación solicitada:
   - `agg=daily`: Se agregan los datos día por día. Para cada fecha dentro del rango, se calcula la temperatura máxima, temperatura mínima, temperatura media (promedio de las medias diarias), el total de precipitación acumulada y el porcentaje medio de nubosidad. El resultado es una lista de días con estos valores agregados (campos `temp_max`, `temp_min`, `temp_avg`, `precip_total_mm`, `cloud_avg_pct`).
   - `agg=rolling7`: Se calcula una *media móvil* de 7 días (aplicable cuando hay al menos 7 días de datos). Para cada día, a partir del séptimo, se calcula el promedio de

temperatura de los últimos 7 días, el promedio de nubosidad de 7 días y la suma de precipitación de 7 días. El resultado es una lista de registros con campos `temp_avg7`, `cloud_avg7_pct` y `precip_sum7_mm`, cada uno asociado a la fecha del último día considerado en esa ventana.

- Sin parámetro `agg`: Si no se solicita ninguna agregación específica, el servicio simplemente devuelve los datos diarios originales ordenados por fecha (equivalente a *datos crudos* por día, tal cual fueron obtenidos del Servicio A).

5. **Conversión de unidades:** Si el parámetro `unit=F`, el Servicio B convierte todas las temperaturas calculadas de grados Celsius a Fahrenheit, aplicando la fórmula $F = C \times \frac{9}{5} + 32$ (con redondeo a 2 decimales para mantener un formato consistente). El campo `unit` en la respuesta indicará claramente "`C`" o "`F`" según la unidad elegida, para que el consumidor interprete correctamente los valores.

6. **Respuesta y almacenamiento en caché:** Finalmente, el Servicio B construye la respuesta JSON consolidada que se enviará al cliente. Esta incluye: la ciudad consultada, la unidad de temperatura, la fecha inicial `from`, la fecha final `to` calculada, y un array `days` con los datos (ya sea crudos o agregados) de cada día. Antes de enviar, almacena esta respuesta en la caché Redis con la clave única mencionada y con un *Time-To-Live* (TTL) configurable (por defecto, 10 minutos). Así, futuras peticiones idénticas dentro de ese período se resolverán directamente desde la caché, mejorando significativamente el rendimiento y reduciendo la carga tanto sobre la base de datos como sobre el Servicio A. La respuesta HTTP incluirá una cabecera `X-Cache` indicando `HIT` o `MISS` para informar si se utilizó la caché.

El Servicio B también cuenta con un endpoint `/health` que devuelve un objeto `{ "status": "ok", "cache": ... }` indicando que el servicio está activo. En el campo `cache` reporta si la caché en uso es Redis (`"redis"`) o, en su defecto, una caché en memoria local (`"memory"`) en caso de que Redis no esté disponible. Este endpoint es útil para monitorear la salud del sistema y se utiliza en el *healthcheck* de Docker Compose.

Cabe mencionar que FastAPI genera automáticamente documentación interactiva para este servicio (disponible en `/docs` con Swagger UI o `/redoc`), facilitando la exploración de la API durante el desarrollo o pruebas. Además, se configuró CORS en el Servicio B para permitir peticiones AJAX desde orígenes locales comunes (por ejemplo, un *frontend* en `localhost:3000` o `5173`), lo cual sería útil de cara a integrar esta API con una interfaz web.

5. Infraestructura y Despliegue

Toda la solución está definida mediante contenedores Docker, con la orquestación de *Docker Compose*. A continuación se resumen los componentes incluidos:

- **Base de datos (PostgreSQL):** Utiliza la imagen oficial `postgres:18-alpine`, con una base de datos llamada `meteo`. En el `docker-compose.yml` se monta un script SQL de inicialización que crea la tabla `weather_readings` con sus columnas y restricciones (ver Sección 3). Los datos se persisten en un volumen Docker para conservarlos entre reinicios. El contenedor expone el puerto 5432 (opcionalmente accesible desde el host para debug) y tiene configurado un *healthcheck* (usando `pg_isready`) para indicar cuándo está listo para aceptar conexiones.

- **Servicio A (C++):** Definido en la sección `servicioa` del Docker Compose. Su *Dockerfile* emplea una compilación multi-etapa: primero construye el binario C++ usando CMake en una imagen *builder* (Debian con las dependencias de desarrollo como `libpqxx-dev`, etc.), y luego lo empaqueta en una imagen de *runtime* mínima (Debian slim) con solo las

bibliotecas necesarias (libpqxx, libpq). El contenedor expone el puerto 8080 y tiene una instrucción de `ENTRYPOINT` que lanza el binario `servicioa` en modo servidor. En tiempo de ejecución, el servicio toma la configuración de conexión a la base de datos de variables de entorno (host, puerto, nombre BD, usuario y contraseña, que Docker Compose provee al contenedor). Además, se definió un *healthcheck* que periódicamente hace una petición HTTP a `/health` en el propio servicio para verificar que sigue operativo y con conexión a la base de datos.

- **Servicio B (Python/FastAPI):** Definido en la sección `serviciob`. Su imagen Docker se basa en `python:3.11-slim` con instalación de dependencias listadas en `requirements.txt` (FastAPI, Uvicorn, httpx, redis, etc.). Se utiliza un entorno virtual dentro del contenedor para aislar las dependencias. El contenedor expone el puerto 8090. Al iniciarse, ejecuta el servidor Uvicorn apuntando a la aplicación FastAPI. Este servicio recibe la URL base del Servicio A (`http://servicioa:8080`) mediante la variable `SERVICE_A_BASE_URL`, y la dirección de Redis mediante `REDIS_URL`, entre otros parámetros configurables (TTL de la caché, opciones de CORS, etc., también mediante variables de entorno). El `depends_on` en Docker Compose asegura que no se lance hasta que Servicio A y Redis estén saludables. Su healthcheck comprueba simplemente que el puerto 8090 está abierto aceptando conexiones TCP.
- **Caché Redis:** Usa la imagen `redis:7-alpine`. Se configura el modo persistente (*append only*) para que las entradas de caché sobrevivan reinicios si fuese necesario. El contenedor `redis` no expone puertos al host (solo es accesible para Servicio B dentro de la red interna). También tiene un *healthcheck* (comando `redis-cli ping`) para indicar disponibilidad.

En el fichero `docker-compose.yml` se definieron dependencias de salud: por ejemplo, Servicio A espera a que el contenedor de base de datos esté *healthy* antes de arrancar, y Servicio B espera tanto a A como a Redis. De esta manera, se garantiza que en el momento en que Servicio B intente acceder a A (o este a la base de datos), los destinos estén operativos, aumentando la robustez en el inicio del sistema. Todos los contenedores están configurados con `restart: unless-stopped` para reanudarlos automáticamente en caso de fallo.

Para desplegar el sistema localmente, basta con tener Docker y Docker Compose instalados. El proceso típico de construcción y arranque es:

```
docker compose up -d --build
```

Este comando construye las imágenes de Servicio A y B, inicia todos los contenedores en segundo plano, y gracias a los healthchecks podemos verificar su estado con:

```
docker compose ps    # cada servicio debería mostrar "Up (healthy)"
```

Una vez todo está en *Up (healthy)*, el sistema está listo para recibir peticiones en los puertos 8080 (Servicio A) y 8090 (Servicio B).

6. Ejemplos de Uso

A continuación se muestran ejemplos representativos de cómo interactuar con la solución mediante `curl`, junto con las respuestas esperadas:

- **Ingesta de datos (Servicio A):** Para cargar un fichero CSV llamado `meteo.csv` en el sistema, se usa el endpoint de ingesta:

  ```
  $ curl -F "file=@meteo.csv" http://localhost:8080/ingest/csv
  ```

  Si la operación es exitosa, el Servicio A devuelve un JSON resumen, por ejemplo:

```
{ "rows_inserted": 3650, "rows_rejected": 2, "elapsed_ms": 1240,
  "file_checksum": "sha256:..." }
```

Este resultado indica que se insertaron 3650 filas nuevas en la base de datos y se rechazaron 2 (por formato inválido o duplicadas), con un tiempo de procesamiento de 1.240 segundos.

- **Consulta meteorológica (Servicio B):** Una vez ingeridos los datos, un cliente puede solicitar información meteorológica agregada. Por ejemplo, el siguiente comando pide datos diarios de Madrid a partir del 15 de octubre de 2025, para 5 días, con las temperaturas en Fahrenheit y agregación diaria:

```
$ curl "http://localhost:8090/weather/Madrid?date=2025-10-15&days=5&unit=F&agg=daily"
```

La respuesta será un JSON con el rango de fechas consultado y una lista de días con los campos agregados. Ejemplo (formato simplificado):

```
{
  "city": "Madrid",
  "unit": "F",
  "from": "2025-10-15",
  "to": "2025-10-19",
  "days": [
    { "date": "2025-10-15", "temp_max": 60.35, "temp_min": 42.53,
      "temp_avg": 51.44, "precip_total_mm": 3.56, "cloud_avg_pct": 65.0 },
    { "date": "2025-10-16", ... },
    ...
  ]
}
```

En este caso (agregación `daily`), para cada día del intervalo se muestran la temperatura máxima, mínima, media, la precipitación acumulada del día (mm) y la nubosidad media (%). El campo `unit = "F"` confirma que las temperaturas han sido convertidas a Fahrenheit. Si en lugar de `agg=daily` se usara `agg=rolling7`, la lista `days` contendría los valores de la media móvil de 7 días hasta cada fecha (campos `temp_avg7`, `cloud_avg7_pct`, `precip_sum7_mm`), proporcionando una visión de tendencias semanales.

(Los ejemplos anteriores suponen que los contenedores están escuchando en `localhost` y que el archivo CSV de ejemplo contiene datos para la ciudad y fechas consultadas. Las respuestas JSON se han formateado para claridad.)

7. Scripts de Prueba y Ejecución

Para facilitar el uso y la validación del sistema, se han creado dos scripts auxiliares que automatizan tareas comunes:

- **`run_tests.sh`**: compila el sistema, ejecuta los tests definidos (unitarios y de integración), y verifica que todos pasen correctamente. Este script está pensado para ser ejecutado en local o en entornos de CI/CD como paso previo a cualquier despliegue.
- **`run_build.sh`**: construye las imágenes Docker necesarias y lanza todos los servicios definidos en el `docker-compose.yml`, asegurando que el sistema quede desplegado y listo para recibir peticiones. Incluye la fase de build y puede utilizarse tanto en desarrollo como en pruebas funcionales.

Ambos scripts están diseñados para simplificar el flujo de trabajo y reducir errores manuales durante el desarrollo o la revisión del proyecto.

8. Decisiones Técnicas y Escalabilidad

Decisiones de Diseño Clave

En el desarrollo de esta solución se tomaron varias decisiones arquitectónicas y tecnológicas relevantes:

- **Tecnología de cada servicio:** Se eligió C++17 para el Servicio A debido a su alto rendimiento en operaciones de parseo de texto y manejo eficiente de memoria, lo que resulta adecuado para procesar potencialmente grandes ficheros CSV. Además, la biblioteca `libpqxx` proporciona una integración robusta con PostgreSQL. Para el Servicio B se empleó Python con FastAPI, privilegiando la rapidez de desarrollo y la claridad para implementar lógica de agregación y manejo de concurrencia de I/O (peticiones async al Servicio A y Redis). FastAPI ofrece documentación automática y un ecosistema rico (por ejemplo, `httpx` para llamadas HTTP asíncronas y `aioredis`), lo que agilizó la implementación de características como los reintentos y la caché.

- **Interfaz de comunicación (HTTP/JSON):** Aunque el enunciado permitía gRPC como alternativa, se optó por HTTP con JSON por simplicidad y familiaridad. JSON facilita la depuración (es legible por humanos) y el uso de herramientas estándar como `curl`. Dado el alcance de la prueba (y la latencia en una red local), HTTP/JSON es suficiente; en un entorno productivo con altísima carga o requisitos de latencia muy bajos, se podría evaluar gRPC, pero no fue necesario en este contexto.

- **Caché con Redis:** Para evitar consultas redundantes al Servicio A (y por ende a la base de datos), se incorporó una capa de caché. Se decidió usar Redis (caché externa) en lugar de una caché en memoria local del Servicio B, principalmente para facilitar una posible escalabilidad horizontal de este servicio: múltiples instancias de Servicio B podrían compartir el mismo Redis y así mantener consistente el cacheado de resultados. Redis, además, ofrece muy buen rendimiento en lectura/escritura y expiración automática de claves. El TTL de 10 minutos fue considerado un equilibrio razonable entre frescura de datos y eficacia de la caché en un escenario de ejemplo (configurable según necesidades reales).

- **Tolerancia a fallos:** Se implementaron mecanismos de reintento exponencial para robustecer la comunicación entre servicios. Esta decisión añade resiliencia: pequeños fallos temporales o reinicios de Servicio A pueden ser sorteados sin impacto inmediato al usuario final (especialmente gracias a la caché, que actúa como *fallback*). Asimismo, los *health-checks* en Docker garantizan la detección y reinicio de contenedores colgados o en mal estado de forma automática.

Escalabilidad y Mejoras Futuras

Aunque el sistema está concebido para la prueba técnica, se tuvieron en cuenta aspectos que facilitan escalarlo o mejorarlo:

- **Escalado horizontal de servicios:** Tanto el Servicio A como el Servicio B podrían replicarse en múltiples instancias detrás de un balanceador de carga para atender un mayor volumen de peticiones. El Servicio B, al ser *stateless* (gracias a que la caché está en Redis), es particularmente fácil de escalar horizontalmente. El Servicio A también puede escalarse; en ese caso, habría que considerar la concurrencia en escrituras a la base de datos, pero PostgreSQL puede manejar múltiples conexiones concurrentes y las restricciones de unicidad seguirían garantizando consistencia.

- **Base de datos:** Para grandes volúmenes de datos o muchas consultas concurrentes, se podría migrar la base de datos hacia un clúster replicado (e.g., usar replicas de solo lectura para distribuir la carga de consultas, manteniendo una primaria para ingestas). También podría evaluarse una base de datos orientada a series temporales o un esquema de particionado por fechas si el historial de muchos años hiciera lenta la consulta por rangos.
- **Mejoras de caché:** El TTL de 10 minutos es un valor fijo de ejemplo; en un escenario real se podría ajustar dinámicamente o invalidar la caché cuando lleguen nuevos datos relevantes (por ejemplo, tras una ingesta que afecte a la ciudad consultada). También podría implementarse un *cache warming* (pre-cargar en caché ciertas consultas frecuentes).
- **Métricas y monitoreo:** Para escalar con confianza sería importante añadir sistemas de monitoreo (por ejemplo, recopilar métricas de tasa de peticiones, latencias, tasas de `cache hit/miss`, uso de CPU/memoria de cada servicio, etc.). Esto ayudaría a detectar cuellos de botella. Por ahora, la solución se centra en la funcionalidad, pero es extensible para incorporar estas herramientas.
- **Paralelismo en ingesta:** Si el archivo CSV de entrada fuese muy grande, se podría paralelizar la validación/inserción por bloques o utilizar COPY de PostgreSQL para inserciones masivas más eficientes. La implementación actual inserta fila a fila en la base de datos (suficiente para volúmenes moderados como el proporcionado en la prueba).
- **Documentación y pruebas:** Se incluyeron especificaciones OpenAPI para ambos servicios (lo que facilita futuras integraciones o generación de clientes automáticos) y pruebas unitarias básicas para las funciones de agregación y conversión. Mantener y ampliar este conjunto de pruebas contribuirá a la calidad del software a medida que evolucione.

En resumen, la solución desarrollada cumple con los requisitos de la prueba técnica, ofreciendo un diseño modular, robusto frente a fallos y con consideraciones de rendimiento. Las decisiones tomadas en cuanto a tecnologías y arquitectura buscan un balance entre eficiencia (en la ingesta y consulta), claridad de implementación y capacidad de crecimiento en el futuro.