**Documentación del Proyecto: Sistema de Análisis Concurrente**

**1. Resumen del Proyecto**

Este proyecto, basado en el desafío "Actividad 3 - Programación concurrente", implementa un sistema de análisis de datos biológicos en tiempo real para la ficticia "Umbrella Corporation".

El objetivo principal es gestionar y procesar concurrentemente múltiples flujos de datos (Genéticos, Bioquímicos y Físicos). El sistema está diseñado para ser altamente eficiente y responsivo, utilizando un **diseño híbrido de concurrencia** que combina asyncio para operaciones de E/S de red (como WebSockets) con multiprocesamiento y multihilo para tareas de cómputo intensivo (CPU) y E/S de disco (bloqueante).

El sistema completo se compone de dos partes principales que se ejecutan simultáneamente:

1. **Backend de Procesamiento (main.py)**: Un servicio en Python que ingiere, normaliza y procesa los datos.
2. **Frontend de Monitorización (web/app.py)**: Un dashboard web que visualiza las métricas y alertas del backend en tiempo real.

**2. Arquitectura y Flujo de Datos**

El sistema sigue un patrón de **Productor-Consumidor** con múltiples etapas, orquestado por asyncio.

1. **Lanzamiento:** El archivo run.py inicia todo. Lanza el backend (main.py) y el servidor web (web/app.py) como dos tareas concurrentes de asyncio.
2. **Ingesta (Productores Primarios):** Los simuladores en ingestion/data\_fetchers.py generan datos crudos (genéticos, bioquímicos, físicos) y los colocan en tres colas de entrada asyncio.Queue separadas (ej. genetic\_input\_queue).
3. **Servicios (Consumidores/Productores Secundarios):** Cada clase de servicio (GeneticoService, BioquimicoService, FisicoService) vigila su cola de entrada correspondiente.
   * **Normalización:** Los datos crudos se limpian y validan usando un Normalizer (ej. GeneticNormalizer).
   * **Métricas:** Se registra el evento en el MetricsCollector.
   * **Alertas:** Se revisan los datos en busca de eventos críticos (ej. "T-VIRUS"). Si se encuentra uno, se llama al AlertManager.
   * **Encolado:** Los datos limpios se colocan en la cola processing\_queue central.
4. **Procesamiento (Consumidor Final):** El DataOrchestrator vigila la processing\_queue.
   * **Delegación de Tareas:** Decide el tipo de tarea.
     + **CPU-Bound** (Genética, Bioquímica): Delega la tarea (ej. analyze\_genetic\_sequence) a un ProcessPoolExecutor para que se ejecute en un núcleo de CPU separado, sin bloquear el programa principal.
     + **I/O-Bound** (Física): Delega la tarea (ej. save\_vitals\_to\_file\_sync) a un ThreadPoolExecutor.
   * **Medición:** Mide el tiempo que tarda la tarea y registra la latencia.
5. **Comunicación Web (Puente):** El DataOrchestrator y el AlertManager envían los resultados (latencias y alertas) a una asyncio.Queue especial llamada data\_queue.
6. **Difusión Web (WebSocket):** Una tarea en segundo plano (websocket\_broadcaster) vigila la data\_queue, saca los mensajes y los difunde a todos los navegadores conectados a la página web.

**3. Componentes del Backend (Código src/)**

**run.py**

Archivo ejecutable principal.

* **Propósito**: Lanzar el sistema.
* **Funcionalidad Clave**:
  + Modifica el sys.path para permitir importaciones limpias desde la carpeta src/.
  + Implementa un "hack" para Windows que establece pythonw.exe como el ejecutable para multiprocessing, evitando que se abran 4 ventanas de consola adicionales.
  + Utiliza asyncio.gather para ejecutar el main (backend) y el run\_web\_server (frontend) de forma concurrente.

**main.py**

El "ensamblador" del backend.

* **Propósito**: Inicializar y conectar todos los componentes del backend.
* **Funcionalidad Clave**:
  + Instancia todos los objetos necesarios: AlertManager, los Normalizers, los Services y el DataOrchestrator.
  + Implementa **Inyección de Dependencias**: Pasa las colas y los managers necesarios a cada clase a través de su constructor (ej. GeneticoService recibe las colas y el AlertManager).
  + Crea las tareas de asyncio para iniciar todos los simuladores de ingesta (simulate\_...\_feed) y los servicios (...\_service.start()).
  + Maneja el apagado ordenado (finally), cancelando todas las tareas.

**config.py**

* **Propósito**: Almacén central de variables de configuración.
* **Variables**:
  + MAX\_CPU\_WORKERS: Número de procesos a usar en el ProcessPoolExecutor.
  + SIMULATION\_SPEED: Multiplicador para la velocidad de la ingesta de datos.

**communication/**

* **Propósito**: Define las "tuberías" que conectan el sistema.
* **Archivos**:
  + queues.py: Instancia las cuatro asyncio.Queue globales: genetic\_input\_queue, biochemical\_input\_queue, physical\_input\_queue (para datos crudos) y processing\_queue (para datos normalizados).

**ingestion/**

* **Propósito**: Simular la llegada de datos del "mundo real".
* **Archivos**:
  + data\_fetchers.py: Contiene tres funciones async (simulate\_genetic\_data\_feed, etc.) que se ejecutan en un bucle infinito, generan datos aleatorios (incluyendo eventos críticos) y los await queue.put() en sus colas respectivas.

**normalization/**

* **Propósito**: Limpiar, validar y estandarizar datos crudos.
* **Archivos**:
  + validators.py:
    - DataNormalizer: Clase base abstracta.
    - GeneticDataModel, BiochemicalDataModel, PhysicalDataModel: Clases de **Pydantic** que definen el *schema* de los datos limpios. Por ejemplo, PhysicalDataModel espera un subject\_id (str) y un heart\_rate (Opcional, int).
    - GeneticNormalizer, BiochemicalNormalizer, PhysicalNormalizer: Clases que implementan la lógica de normalize(). Usan model\_validate() de Pydantic para forzar los datos crudos a coincidir con el *schema*, limpiando formatos (ej. "98%" -> 98) en el proceso.

**services/**

* **Propósito**: Orquestar el primer paso del procesamiento: Ingesta, Normalización y Alerta.
* **Archivos**:
  + base\_service.py: Clase BaseDataService (ABC). Define el \_\_init\_\_ (que almacena las colas, el normalizador y el alert manager) y la lógica principal \_process\_data y start.
  + genetico\_service.py, bioquimico\_service.py, fisico\_service.py: Clases concretas que heredan de BaseDataService. Su única responsabilidad es implementar el método \_check\_for\_critical\_events con la lógica de negocio específica (ej. FisicoService comprueba si heart\_rate > MAX\_HEART\_RATE).

**processing/**

* **Propósito**: El "motor" del sistema. Ejecuta el cómputo pesado de forma concurrente.
* **Archivos**:
  + orchestrator.py: Contiene la clase DataOrchestrator.
    - \_\_init\_\_: Inicializa los ProcessPoolExecutor (para CPU) y ThreadPoolExecutor (para E/S).
    - \_route\_and\_process\_task: El método clave. Obtiene un trabajo, mide el start\_time, y usa loop.run\_in\_executor() para enviar la función de cómputo (ej. cpu\_tasks.analyze\_genetic\_sequence) al *pool* de procesos correcto.
    - Al finalizar, mide la duration\_ms y la envía a la data\_queue del WebSocket.
  + cpu\_tasks.py: Contiene las funciones de cómputo pesado (analyze\_genetic\_sequence, analyze\_biochemical\_model). Estas simulan trabajo intensivo de CPU (sin asyncio.sleep) para justificar el uso de ProcessPoolExecutor.
  + io\_tasks.py: Contiene las funciones de E/S. save\_vitals\_to\_file\_sync usa time.sleep(0.5) para simular una escritura a disco bloqueante, justificando el ThreadPoolExecutor.

**monitoring/**

* **Propósito**: Servir como el "marcador" central y seguro para hilos.
* **Archivos**:
  + metrics.py: Define la clase MetricsCollector como un **Singleton**. Usa threading.Lock() para garantizar que solo se cree una instancia.
  + Proporciona métodos thread-safe (usando with self.events\_lock:) para registrar eventos, errores y latencias.
  + get\_current\_stats(): Devuelve una instantánea de todas las métricas, calculando los promedios.

**alerting/**

* **Propósito**: Gestionar el envío de alertas.
* **Archivos**:
  + alert\_manager.py: Define AlertManager. El método send\_alert es llamado por los Services.
  + Imprime la alerta en la consola.
  + Envía la alerta a la data\_queue del WebSocket para que se muestre en la web.
  + Registra la latencia de la alerta en el MetricsCollector.

**4. Componentes del Frontend (Código src/web/)**

**app.py**

* **Propósito**: El servidor web **FastAPI**.
* **Funcionalidad Clave**:
  + @app.on\_event("startup"): Inicia la tarea websocket\_broadcaster.
  + @app.get("/"): Sirve el archivo index.html.
  + @app.get("/api/metrics"): Endpoint de API que devuelve los contadores totales desde MetricsCollector.get\_current\_stats().
  + @app.websocket("/ws"): Acepta y gestiona las conexiones WebSocket de los navegadores.

**connection\_manager.py**

* **Propósito**: El "puente" de comunicación en vivo entre el backend y el frontend.
* **Componentes**:
  + ConnectionManager: Una clase que mantiene una lista de todos los clientes WebSocket (active\_connections).
  + data\_queue: Una asyncio.Queue global. El backend (Orchestrator, AlertManager) *coloca* mensajes en esta cola.
  + websocket\_broadcaster: Una tarea async que *obtiene* mensajes de la data\_queue y los difunde (manager.broadcast) a todos los clientes web conectados.

**templates/index.html**

* **Propósito**: La estructura HTML del dashboard.
* **Funcionalidad Clave**:
  + Define las tarjetas (cards) para las métricas (card-total-processed).
  + Define el <canvas id="realTimeLatencyChart"> para el gráfico de Chart.js.
  + Define la lista id="alert-list" para las alertas.
  + Importa las bibliotecas de Chart.js (localmente) y el dashboard.js.

**static/css/style.css**

* **Propósito**: Dar estilo al dashboard.
* **Funcionalidad Clave**:
  + Define un tema oscuro "Umbrella" (--bg-color: #1a1a1d;).
  + Usa CSS Grid (.dashboard-grid) para la maquetación.
  + Define la altura fija (height: 450px) para el contenedor del gráfico para evitar que se expanda verticalmente.
  + Asegura que el gráfico y las alertas ocupen todo el ancho (grid-column: 1 / -1;).

**static/js/dashboard.js**

* **Propósito**: Lógica del cliente (navegador).
* **Funcionalidad Clave**:
  + **Gráfico**: Inicializa realTimeLatencyChart de Chart.js con tres *datasets* (Genetic, Biochemical, Physical) y una escala de tiempo (type: 'time').
  + **Conexión WebSocket**: Llama a connectWebSocket(), que se conecta al endpoint /ws.
  + ws.onmessage: Es el receptor principal. Escucha los mensajes, los parsea (JSON.parse) y usa un switch para decidir si es "latency" (llama a addDataToChart) o "alert" (llama a addAlertToList).
  + **Conexión Fetch**: Llama a updateAggregateMetrics() cada 2 segundos (setInterval), que hace fetch al endpoint /api/metrics para actualizar los contadores totales.
  + addDataToChart: Añade el nuevo punto de latencia y null a los otros *datasets*. Usa spanGaps: true (configurado en el gráfico) para asegurar que las líneas sean continuas.

**5. Pruebas (tests/)**

El proyecto incluye un conjunto de pruebas unitarias para validar la lógica central:

* test\_normalization.py: Prueba que los Normalizers limpien y rechacen datos correctamente (ej. "85.5 ppm" se convierte en 85.5).
* test\_monitoring.py: Confirma que MetricsCollector es un **Singleton** ( m1 is m2 ) y que los contadores y promedios se calculan correctamente.
* test\_services.py: (Archivo test\_services.py en realidad contiene tests para processing, lo cual es un error en el nombre del archivo).
* test\_processing.py: (Archivo test\_processing.py en el repositorio) Usa pytest-mock para simular (mock) los *pools* de ejecutores. Verifica que el DataOrchestrator enruta correctamente las tareas "genetic" al cpu\_executor y las "physical" al io\_executor.