Manifiesto de Ingeniería: Astroflora v5.0 "Antares"

Para: El Equipo de Astroflora
De: Arquitecto Jefe
Asunto: El Plano Maestro de Nuestra Arquitectura
Este documento es nuestra única fuente de verdad. Es el plano de la estación espacial que estamos construyendo. Cada decisión aquí está diseñada para un propósito: crear un sistema que no solo funcione, sino que sea **antifrágil**. Un sistema que se fortalece con el estrés, que es predecible en su comportamiento y un placer de evolucionar.

1. La Filosofía: Construir la Estación Espacial

No estamos construyendo una aplicación web. Estamos construyendo un laboratorio científico autónomo en la nube. Nuestra filosofía de ingeniería debe reflejar esa ambición.

- **Componentes Modulares, no un Monolito:** Cada pieza de nuestra estación (un servicio) es un módulo autónomo con una única función. No construimos un único bloque gigante.
- **Puertos de Conexión Estándar, no Cables Soldados:** Los módulos se comunican a través de interfaces bien definidas (nuestros "contratos"). Esto nos permite actualizar o reemplazar un módulo sin que el resto de la estación se dé cuenta.
- **Ensamblaje Centralizado, no Construcción Anárquica:** Todos los módulos son construidos y conectados en una "fábrica" central (nuestro Contenedor de Dependencias). Esto garantiza la consistencia y elimina las dependencias ocultas.
- **Sistemas de Soporte Vital, no un Lujo:** La resiliencia, la seguridad y la observabilidad no son características que "añadimos después". Son el soporte vital de la estación, integrados en el núcleo desde el primer día.

2. La Estructura de Carpetas Definitiva

Esta es la geografía de nuestro proyecto. Cada archivo y cada carpeta tiene un propósito claro. Esta estructura no es una sugerencia, es la ley. La seguiremos desde el día uno para garantizar la coherencia y la escalabilidad.

```
astroflora_v5/
│
└── src/
    │
    ├── main.py               # El interruptor principal. Inicia la API y el ciclo de vida.
    ├── container.py          # LA FÁBRICA. El corazón de la arquitectura. Construye y conecta todo.
    │
    ├── api/                  # LA TORRE DE CONTROL. Habla con el mundo exterior (Endpoints HTTP).
    │   ├── dependencies.py   # Guardias de seguridad de la API (autenticación, etc.).
    │   └── routers/          # Agrupación de endpoints por función (análisis, usuarios, etc.).
    │
    ├── config/               # EL PANEL DE CONTROL. Configuración centralizada.
    │   └── settings.py
    │
    ├── core/                 # EL PUENTE DE MANDO. La lógica de negocio de alto nivel.
    │   ├── orchestrator.py   # El Comandante de la Misión. Dirige el flujo.
    │   └── pipeline.py       # El Plan de Vuelo. Define los pasos de un análisis científico.
    │
    ├── models/               # LOS PLANOS. Define la estructura de todos nuestros datos.
    │   ├── dto.py            # Data Transfer Objects (Pydantic) para la API.
    │   └── orm.py            # Object-Relational Mapping (SQLAlchemy) para la base de datos.
    │
    ├── services/             # LA SALA DE MÁQUINAS Y EQUIPOS.
    │   ├── interfaces.py     # LOS PUERTOS DE CONEXIÓN UNIVERSAL. Contratos para cada servicio.
    │   │
    │   ├── execution/        # El sistema de transporte de trabajos.
    │   │   └── sqs_dispatcher.py
    │   │
    │   ├── observability/    # La cubierta de telemetría.
    │   │   └── metrics_service.py
    │   │
    │   ├── resilience/       # El blindaje y los sistemas de soporte vital.
    │   │   ├── capacity_manager.py
    │   │   └── circuit_breaker.py
    │   │
    │   └── security/         # El puesto de control de seguridad.
    │       └── etdi_verifier.py
    │
    └── workers/              # LOS LABORATORIOS AUTÓNOMOS.
        └── analysis_worker.py  # El proceso que realmente ejecuta el análisis científico.
```

3. El Flujo de la Misión: Anatomía de una Solicitud de Análisis

Para entender cómo opera "Antares", sigamos el viaje de una única solicitud de análisis a través del sistema.

ESCENARIO: Una científica envía una secuencia de proteína para ser analizada.

1. **LLEGADA A LA TORRE DE CONTROL (api/routers/analysis.py)**
   - La solicitud HTTP POST llega a nuestro endpoint /api/v1/analysis.
   - Un "guardia de seguridad" (api/dependencies.py) intercepta la llamada, verifica el token JWT del usuario y se asegura de que tenga permiso para estar allí.
2. **CONSULTA AL GESTOR DE CAPACIDAD (services/resilience/capacity_manager.py)**
   - El endpoint no procesa nada todavía. Primero, pregunta al CapacityManager: "¿Tenemos capacidad para un nuevo trabajo?".
   - El CapacityManager consulta Redis para ver cuántos trabajos se están ejecutando.
   - **Si hay capacidad:** El flujo continúa.
   - **Si no hay capacidad:** El CapacityManager añade la solicitud a una lista de espera (en Redis) y la API responde inmediatamente con un error 429 Too Many Requests. La científica sabe que el sistema está ocupado y su solicitud está en cola. Esto protege al sistema de la sobrecarga.
3. **EL COMANDANTE TOMA EL CONTROL (core/orchestrator.py)**
   - El endpoint pasa la solicitud al IntelligentOrchestrator.
   - El Comandante no hace el trabajo pesado. Su misión es orquestar.
   - **Acción 1:** Crea un registro en la base de datos (AnalysisContext) para rastrear este análisis, marcándolo con el estado QUEUED.
   - **Acción 2:** Llama al sistema de transporte de trabajos.
4. **DESPACHO AL SISTEMA DE TRANSPORTE (services/execution/sqs_dispatcher.py)**
   - El Orquestador le dice al SQSDispatcher: "Toma este ID de análisis y ponlo en la cola de trabajos".
   - El SQSDispatcher crea un mensaje con el context_id y lo envía a una cola de mensajes de AWS (SQS).
   - **PUNTO CLAVE:** La API responde 202 Accepted a la científica en este momento. La respuesta es casi instantánea. El trabajo pesado se hará en segundo plano.
5. **EL LABORATORIO AUTÓNOMO SE ACTIVA (workers/analysis_worker.py)**
   - En un lugar completamente separado (una Lambda de AWS o un contenedor),

un "worker" está constantemente escuchando la cola SQS.
    - Recibe el mensaje con el context_id.
    - Ahora, el worker le dice al IntelligentOrchestrator: "Tengo este trabajo. Ejecuta el plan de vuelo".
6. **EJECUCIÓN DEL PLAN DE VUELO (core/pipeline.py)**
    - El Orquestador carga el LaboratoryPipeline correspondiente. Este plan define los pasos: Paso 1: BLAST, Paso 2: UniProt, Paso 3: Resumen con LLM.
    - El Orquestador actualiza el estado del análisis en la DB a PROCESSING.
    - Itera sobre cada paso del plan. Para cada paso (ej. BLAST):
        - **a. Verificación de Seguridad (ETDIVerifier):** Antes de llamar al servicio BLAST, pregunta al ETDIVerifier: "¿Es este servicio BLAST legítimo y no ha sido manipulado?". El verificador comprueba la firma criptográfica del servicio.
        - **b. Blindaje Activado (CircuitBreaker):** La llamada al servicio BLAST está envuelta en un CircuitBreaker. Si el servicio BLAST ha estado fallando repetidamente, el circuito estará "abierto" y la llamada fallará instantáneamente sin esperar, protegiendo a nuestro sistema.
        - **c. Llamada al Servicio Externo:** Se realiza la llamada HTTP real.
        - **d. Telemetría (MetricsService):** El resultado (éxito/fallo) y la duración de la llamada se registran en nuestro MetricsService.
7. **FIN DE LA MISIÓN**
    - Una vez que todos los pasos del pipeline se completan, el Orquestador guarda el resultado final en la base de datos.
    - Actualiza el estado del análisis a COMPLETED (o FAILED si algo salió mal).
    - Le dice al CapacityManager que un trabajo ha terminado, liberando un espacio para el siguiente de la lista de espera.

4. El Rol de Cada Componente: Desglose Técnico

- **container.py - El Director de la Fábrica:**
    - **Rol:** Es el único lugar en todo el código que crea instancias de nuestros servicios (new PrometheusMetricsService(), new RedisCapacityManager(...)).
    - **Lógica:** Cuando se inicia la aplicación, el contenedor se construye a sí mismo. Lee las dependencias de cada servicio (ej. "el Orchestrator necesita un ISQSDispatcher") y se las "inyecta" en el constructor. Esto se llama **Inyección de Dependencias (DI)**. Garantiza que los componentes estén débilmente acoplados.
- **services/interfaces.py - Los Puertos de Conexión Universal:**
    - **Rol:** Definen los "contratos" o "enchufes". Un Protocol en Python como ICapacityManager simplemente dice: "Cualquier clase que quiera gestionar la

capacidad debe tener un método can_process_request() -> bool".
    - **Lógica:** No tienen código, solo definiciones. Permiten que el Orchestrator dependa de ICapacityManager, sin saber ni importarle si la implementación usa Redis, una base de datos o magia. Esto nos da una flexibilidad extrema para cambiar o probar componentes.
- **core/orchestrator.py - El Comandante de la Misión:**
    - **Rol:** Tomar decisiones de alto nivel. Su código es limpio y legible porque delega todo el trabajo pesado.
    - **Lógica:** Su lógica es un flujo de trabajo: if capacity.can_process() then dispatcher.dispatch(). No contiene lógica de httpx, redis o boto3. Solo habla con las interfaces de los servicios que le fueron inyectados por el contenedor.
- **services/resilience/ - El Blindaje y Soporte Vital:**
    - **CircuitBreaker:** Mantiene un contador de fallos para un servicio externo (en Redis). Si los fallos superan un umbral, "abre el circuito" y rechaza nuevas llamadas a ese servicio por un tiempo, evitando fallos en cascada.
    - **CapacityManager:** Usa un contador y una lista en Redis para rastrear los trabajos activos y encolar las nuevas solicitudes cuando el sistema está al máximo de su capacidad.
- **workers/analysis_worker.py - El Laboratorio Autónomo:**
    - **Rol:** Es un proceso completamente separado de la API. Puede ser una función de AWS Lambda que se activa por un mensaje en SQS.
    - **Lógica:** Su lógica es simple: 1. Recibe un mensaje de SQS. 2. Extrae el context_id. 3. Pide al Orchestrator que ejecute el pipeline para ese contexto. 4. Elimina el mensaje de la cola. Al estar separado, podemos escalar los workers independientemente de la API. Si necesitamos procesar 1000 análisis a la vez, simplemente ejecutamos 1000 instancias de este worker.
- **api/ - La Torre de Control:**
    - **Rol:** Ser la cara del sistema al mundo exterior. Debe ser rápida, segura y estúpida.
    - **Lógica:** Su única lógica es: 1. Autenticar la petición. 2. Validar los datos de entrada (gracias a Pydantic). 3. Pasar la solicitud al Orchestrator. 4. Devolver una respuesta inmediata. Nunca debe realizar trabajo pesado o de larga duración.

Este diseño, Luis, es el resultado de nuestra evolución. Es la síntesis de todas nuestras lecciones. Es simple, pero no simplista. Es robusto, flexible y está listo para cambiar el mundo.