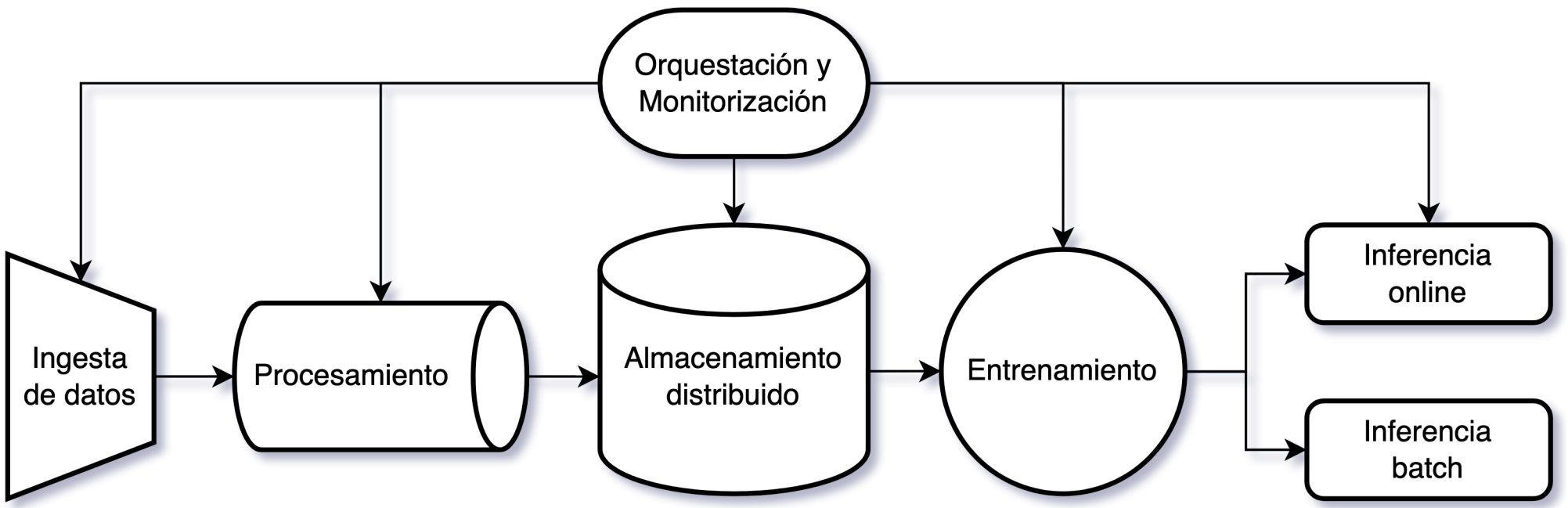
Arquitecturas distribuidas

Ingeniería del Software para Inteligencia Artificial

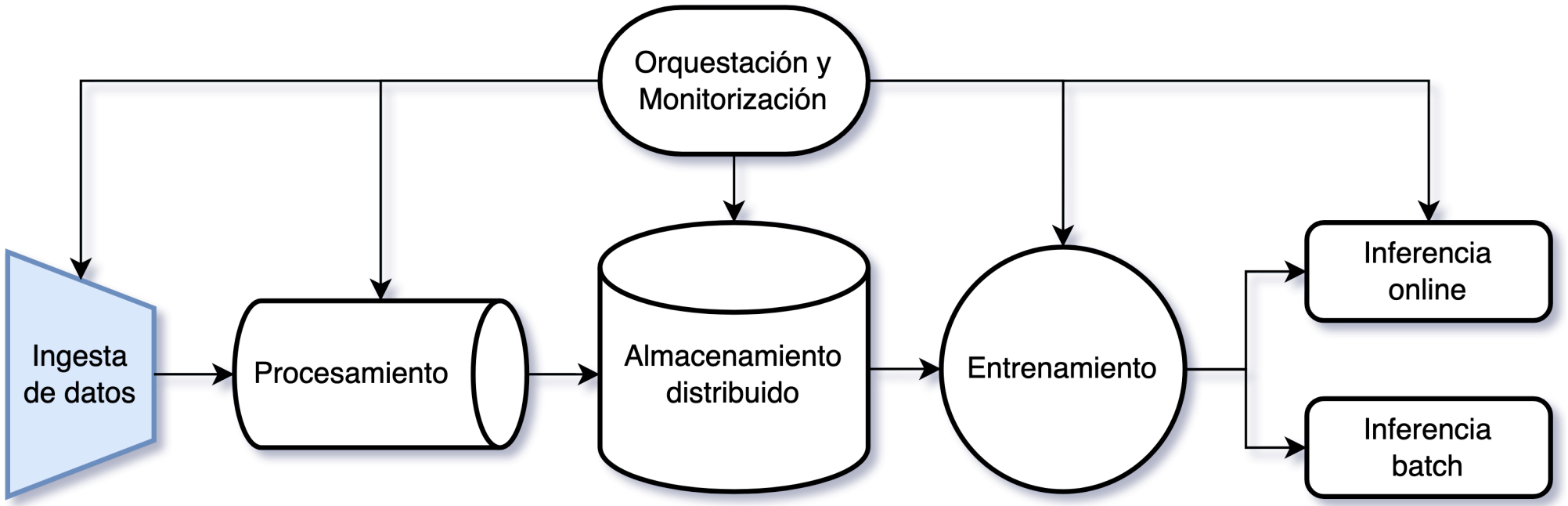
¿Por qué distribuir una arquitectura?

- Hay varios motivos que obligan a la distribución:
  - Volumen de datos
  - Necesidades de rendimiento
  - Diversidad de componentes
- **Las arquitecturas distribuidas son la norma, no la excepción.**

Componentes típicos



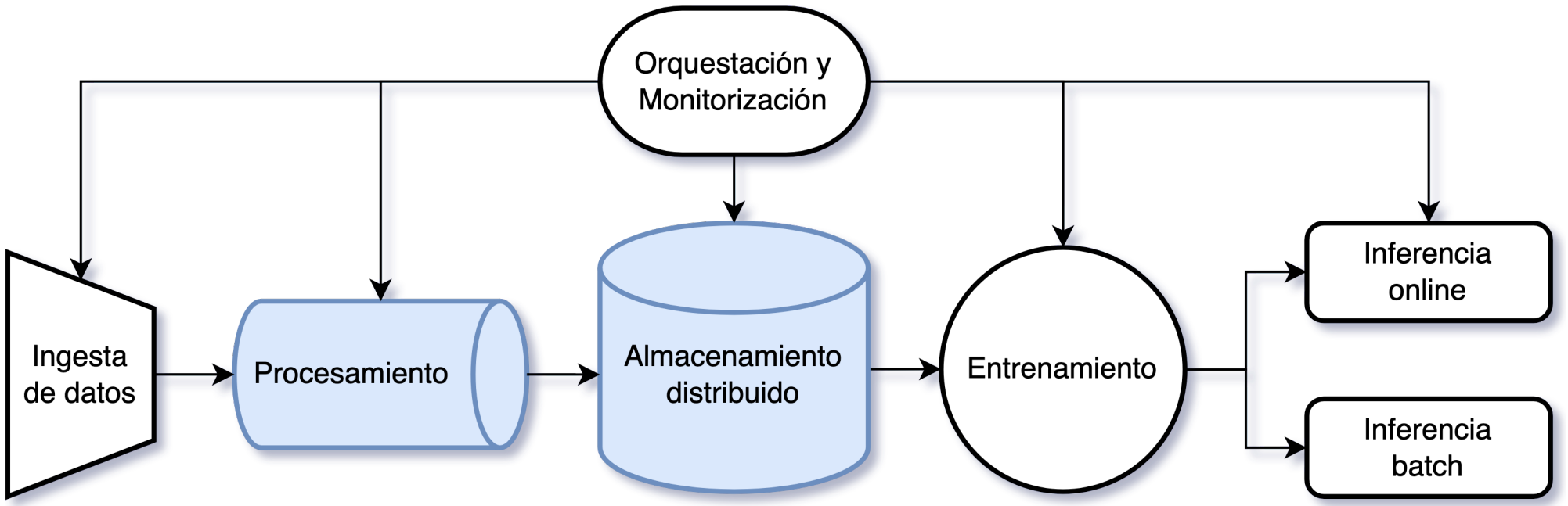
Componentes típicos



Ingesta de datos

- Captura de logs, sensores, eventos, etc.
- Puede ser streaming (Kafka, Kinesis) o batch (ETL).

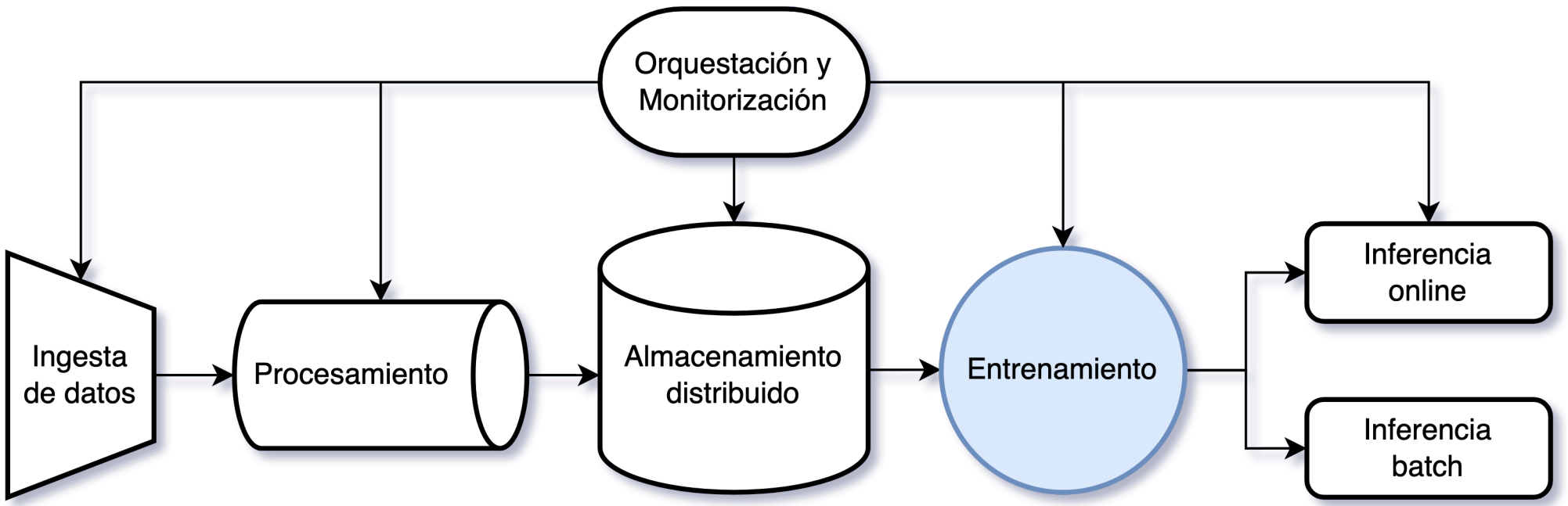
Componentes típicos



Procesamiento y almacenamiento

- Limpieza, transformación, almacenamiento en lagos o almacenes distribuidos.

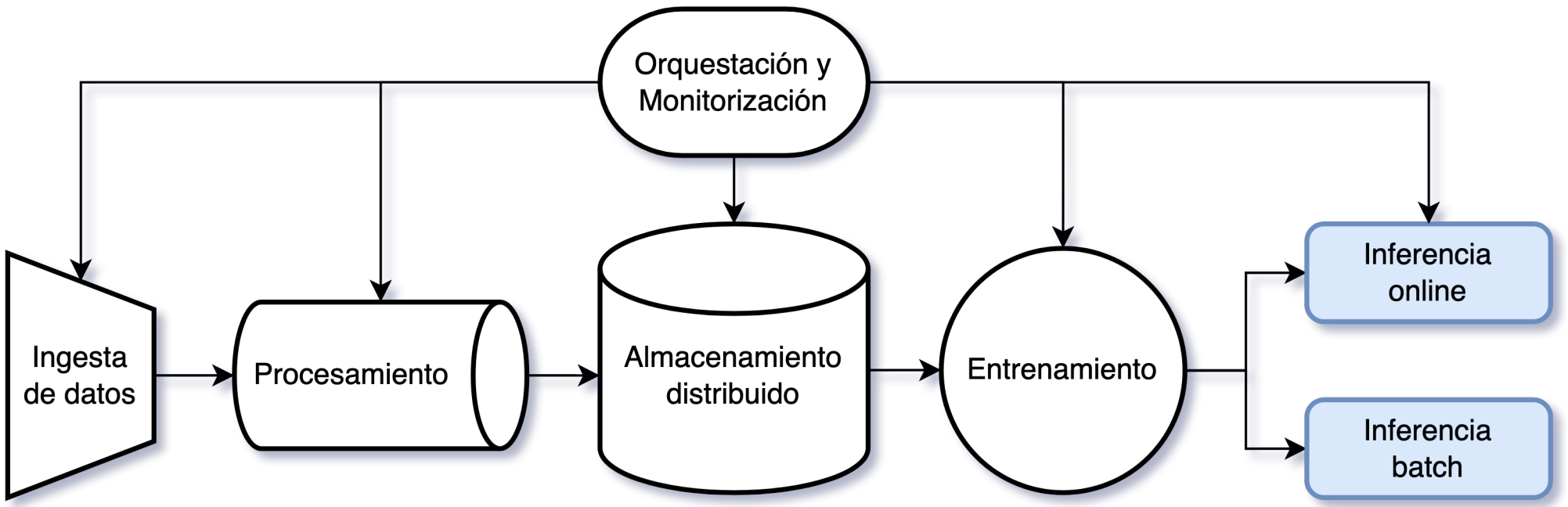
Componentes típicos



Entrenamiento

- Consumo intensivo de cómputo y datos (clusters, autoscaling).
- A menudo ejecutado en pipelines asincrónos.

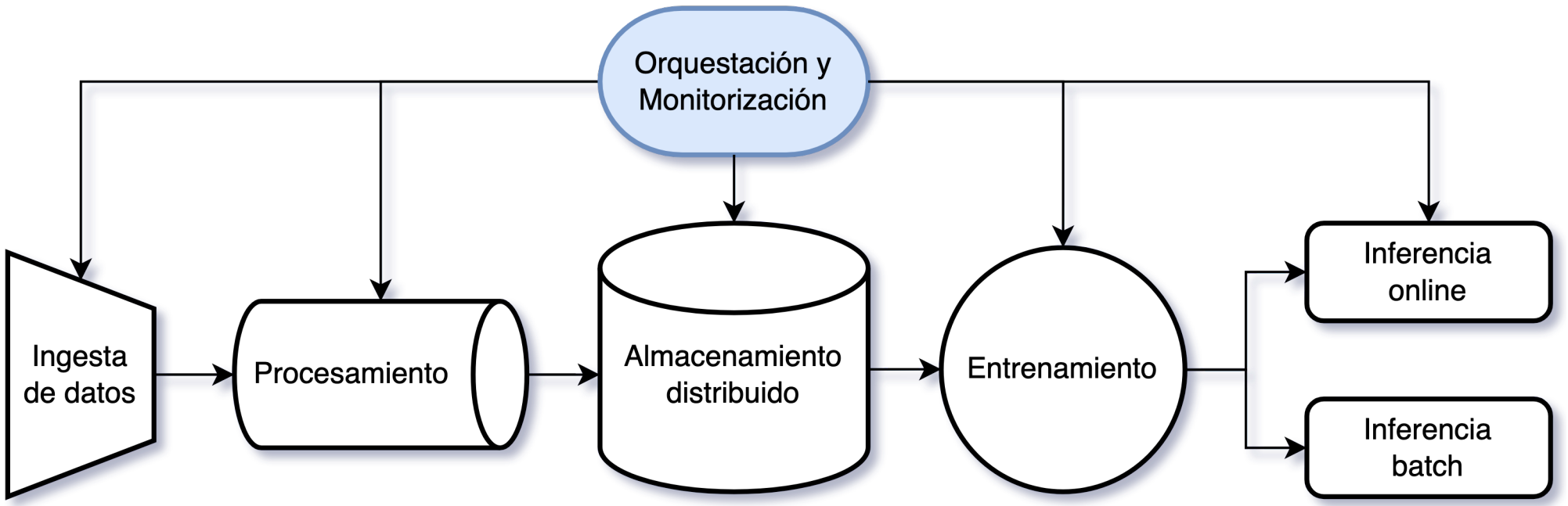
Componentes típicos



Inferencia

- **Online (síncrona):** necesita respuesta en tiempo real.
- **Batch (asíncrona):** recomendaciones, informes, etc.

Componentes típicos



Orquestación y monitoreo

- Airflow, Kubeflow, MLFlow, Prometheus, etc.

¿Por qué distribuir estos componentes?

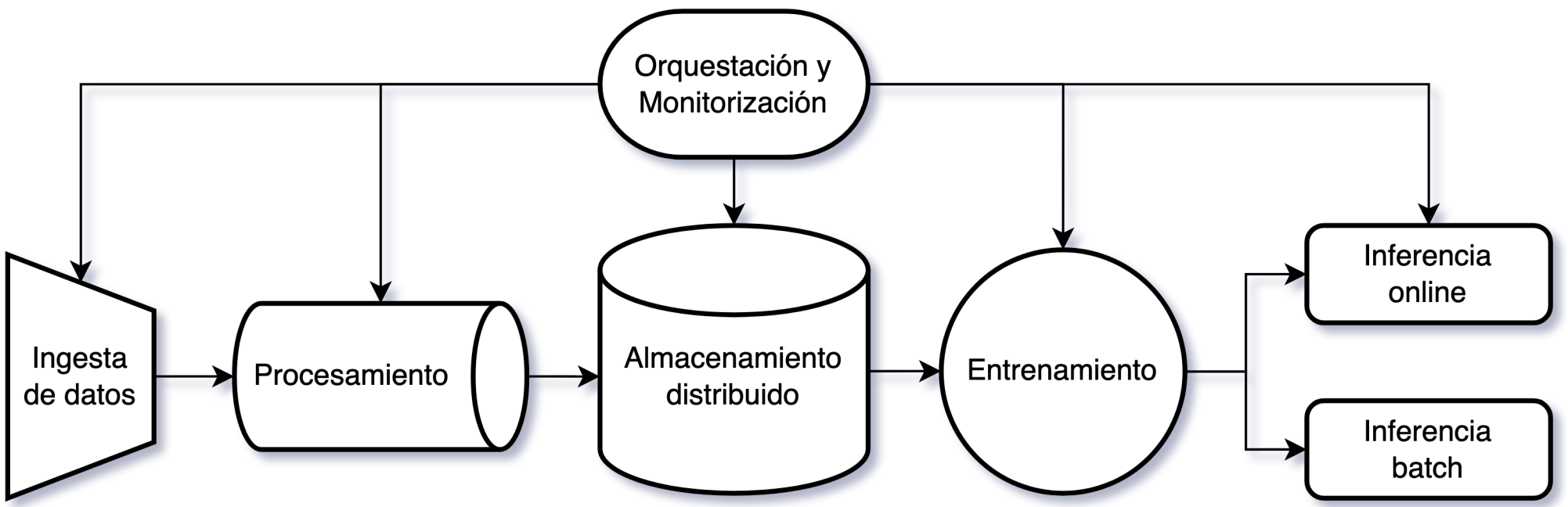
Necesidad	Motivo de Distribución
Escalabilidad	Para manejar millones de peticiones o datos simultáneos
Latencia	Colocar nodos cerca del usuario final (edge, CDN, etc.)
Especialización	Cada componente puede usar el mejor stack/hardware
Tolerancia a fallos	Separar módulos evita que un fallo derribe todo el sistema
Costo	Uso eficiente de recursos (servidores on-demand, spot, etc.)

Comunicación entre componentes

Comunicación	Casos típicos	Ventajas	Desafíos
Síncrona	Inferencia online, APIs	Simple, respuesta inmediata	Acoplamiento, menos resiliente
Asíncrona	Entrenamiento, pipelines de datos	Escalable, desacoplado	Latencia, complejidad

¿Cuál de estos componentes es más difícil de distribuir?

¿Por qué?



Caso de estudio:

Arquitectura de Personalización en Netflix

Arquitectura de Personalización en Netflix

- Netflix utiliza una arquitectura distribuida para personalizar la experiencia de cada usuario de forma escalable y en tiempo real.
- Combina procesamiento masivo en batch con servicios online de inferencia, y se apoya fuertemente en microservicios y almacenamiento compartido de características.

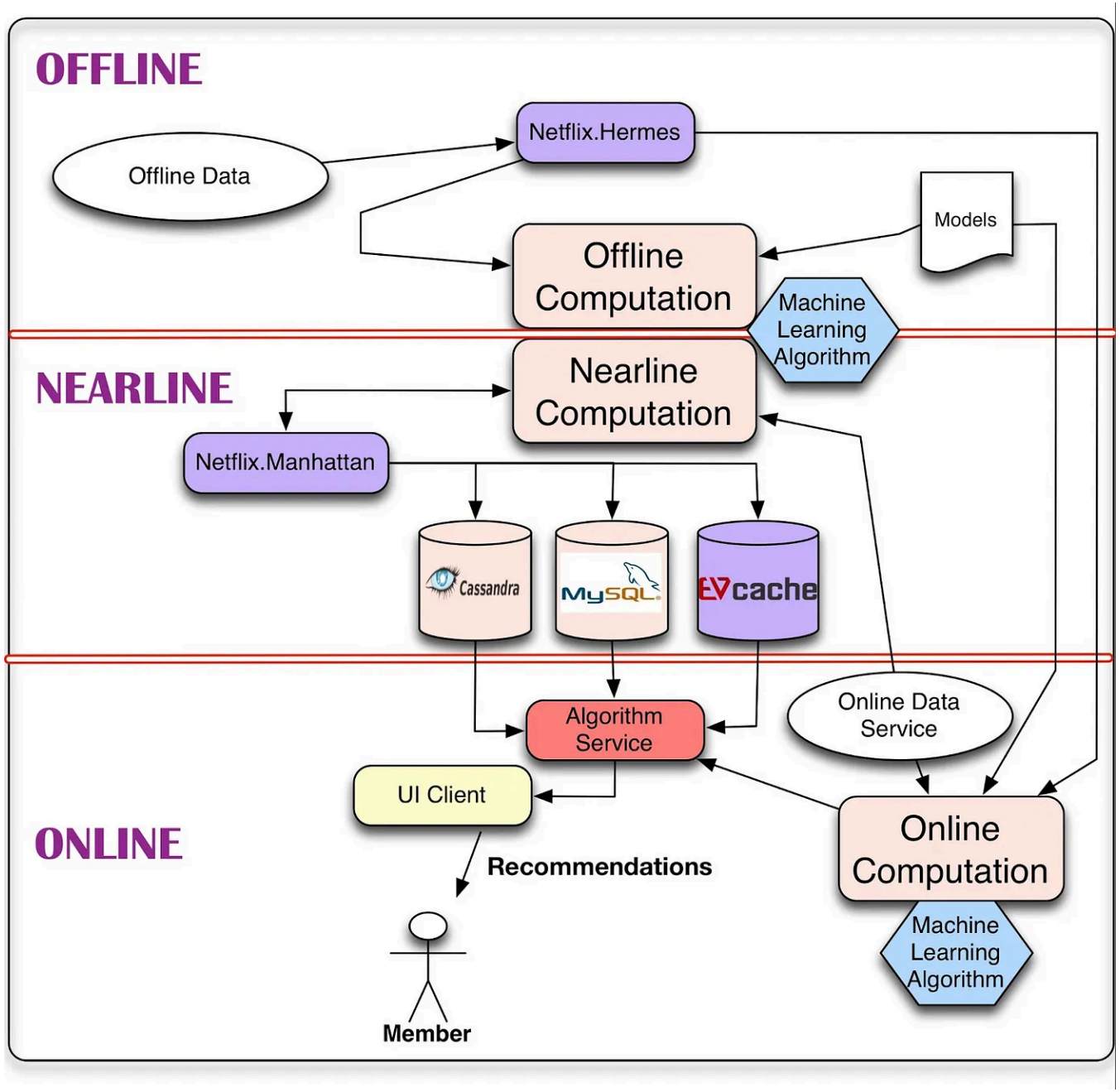
Fuente de información: Blog de Netflix

Objetivos principales:

- Mostrar contenido relevante para cada usuario.
- Hacerlo con baja latencia, alta disponibilidad y precisión.
- Escalar para cientos de millones de usuarios globalmente.

Preguntas:

- ¿Qué tiempo de respuesta es aceptable para el usuario?
- ¿Es necesario procesar los datos que generan los usuarios en tiempo real?


OFFLINE
Offline Data
Netflix.Hermes
Models
Offline Computation
Machine Learning Algorithm
NEARLINE
Nearline Computation
Netflix.Manhattan
Cassandra
MySQL
EVcache
Algorithm Service
Online Data Service
UI Client
ONLINE
Recommendations
Online Computation
Machine Learning Algorithm
Member

Offline

Procesamiento batch clásico, **completamente desacoplado del tiempo real**, usado para entrenamiento, análisis y generación de artefactos pesados.

Características:

- Latencia: horas a días.
- Permite entrenar modelos complejos sin restricciones de tiempo.

Ejemplo: entrenar nuevos modelos de recomendación.

Nearline

Procesamiento casi en tiempo real, **asíncrono**.

Características:

- Latencia: minutos a horas.
- Balance entre latencia y carga computacional.
- Se usa para actualizar modelos intermedios o *features* que cambian con frecuencia.

Ejemplo: generar *scores* para contenido recién añadido.

Online

Procesamiento en tiempo real, **síncrono** con la interacción del usuario.

Características:

- Latencia: milisegundos a pocos cientos de ms.
- Se ejecuta al momento de una solicitud del cliente (por ejemplo, abrir la app).
- Imprescindible para una experiencia fluida.

Ejemplo: generar una lista personalizada de títulos al abrir la pantalla de inicio.

¿Qué componentes se deben distribuir?

Netflix identifica varios componentes clave en su arquitectura de recomendación:

- **Recolección de datos**
- **Procesamiento y almacenamiento**
- **Entrenamiento de modelos**
- **Inferencia y servicio**
- **Evaluación y experimentación**

Criterios para distribuir los componentes

1. Latencia

- **Alta latencia tolerable:** se puede ejecutar en procesos batch o asíncronos (entrenamiento, *feature generation*).
- **Latencia crítica:** se debe servir desde APIs rápidas, cercanas al usuario (inferencia online).

Ejemplo:
Embeddings precalculados en batch → consultados en tiempo real por modelos online.

2. Frecuencia de actualización

- **Datos que cambian lentamente (por ejemplo, características del contenido):** se pueden calcular y almacenar con menor frecuencia.
- **Datos altamente dinámicos (comportamiento de usuario):** requieren actualizaciones frecuentes o cálculos en tiempo real.

Ejemplo:
Interacción reciente del usuario influye en el ranking personalizado de títulos al abrir la app.

3. Escalabilidad

- Componentes con procesamiento intensivo (como el entrenamiento) se distribuyen en clústeres grandes, escalando por lotes.
- Componentes con muchas llamadas pequeñas (inferencias) se escalan horizontalmente como microservicios.

Ejemplo: inferencia distribuida por región.

4. Aislamiento y resiliencia

- Los sistemas se distribuyen para evitar puntos únicos de fallo.
- Microservicios desacoplados facilitan degradación controlada del sistema.

Pregunta:

¿Qué se debería hacer si falla el sistema de personalización?

Caso de estudio:

Tesla *Full Self-Driving* (FSD)

Arquitectura distribuida: vehículo/servidor

Tesla adopta una arquitectura **edge-first**

- Los vehículos realizan **toda la inferencia en tiempo real a bordo**
- Los servidores en la nube se encargan del **entrenamiento, validación y actualización** de los modelos.

Componentes en el vehículo (*edge*)

- **Percepción en tiempo real:** identifican objetos, líneas de carril, semáforos, peatones, etc.
- **Fusión de sensores y predicción:** predicen trayectorias de objetos dinámicos.
- **Control:** seleccionan la mejor acción según la predicción del entorno (frenar, girar, acelerar).
- **Logging y triggers de eventos:** no todo se sube, se usan *triggers inteligentes* para decidir qué eventos recopilar.

Componentes en los servidores (backend)

- **Recolección y priorización de datos:** se almacenan, agrupan y priorizan por tipo, rareza o relevancia.
- **Autoetiquetado:** reconstruyen las escenas en 3D y etiquetan los objetos.
- **Entrenamiento distribuido:** se entrenan redes neuronales masivas usando clústeres de GPU.
- **Validación y simulación:** los modelos se prueban en entornos simulados con datos reales.
- **Despliegue OTA (Over-The-Air)**: modelos actualizados se envían de forma remota a toda la flota.

Claves de diseño

- **Inferencia 100% local:** permite autonomía completa sin conectividad.
- **Aprendizaje federado implícito:** cada coche contribuye datos útiles sin necesitar supervisión constante.
- **Ciclo cerrado:** los datos reales mejoran el modelo, que luego mejora el coche.

Pregunta: ¿Qué ventajas y desventajas tiene hacer inferencia 100% local?

Caso de estudio

OpenAI / ChatGPT

Descripción general

- OpenAI opera modelos de lenguaje a gran escala, como ChatGPT, que requieren arquitecturas distribuidas tanto para su entrenamiento como para su inferencia.
- La naturaleza de estos modelos impone exigencias extremas en términos de rendimiento, disponibilidad y eficiencia operativa.

Características clave de la arquitectura

- **Inferencia distribuida (online):** las solicitudes de los usuarios se enrutan dinámicamente a nodos de inferencia disponibles.
- **Entrenamiento masivo (offline, asíncrono):** entrenamiento distribuido en superclusters de GPUs, usando paralelismo de datos y de modelo.
- **Flujo de datos y feedback:** logs de uso, feedback del usuario y métricas se recopilan de forma asíncrona.

Tipo de comunicación entre componentes

Componente	Tipo de comunicación	Justificación
Cliente ↔ Inferencia	Síncrona	Necesita respuesta inmediata al usuario
Ingesta de logs	Asíncrona	No crítico en tiempo real
Entrenamiento	Mixta (interna)	Coordinación síncrona entre nodos, pero asíncrona respecto al sistema externo

Pregunta: ¿Cómo se establece la comunicación entre el cliente y el modelo de inferencia?

Otros sistemas a explorar

- **Uber Eats:** predicción de tiempo de entrega.
- **Spotify:** recomendaciones de música.
- **Amazon Go:** tiendas sin cajeros.
- **Stripe Radar:** detección de fraudes.
- **Airbnb:** precios dinámicos.
- **Duolingo:** personalización de ejercicios.

Diseña tu propia arquitectura

Contexto del producto

- **MediAssist AI** es una startup pequeña que desarrolla una herramienta para **asistir a médicos de atención primaria** durante las consultas.
- El objetivo es ayudar a tomar decisiones clínicas más rápidas y basadas en evidencia.

Características del producto

- **Sugerencias de diagnóstico** basadas en síntomas y antecedentes.
- **Recomendación de pruebas o exámenes** según guías clínicas.
- **Advertencias en tiempo real** ante posibles interacciones entre medicamentos.
- **Resumen automático de la consulta** para el historial del paciente.

Objetivo del sistema de IA

Proveer soporte clínico inteligente que:

- Responda en tiempo real durante la consulta (latencia baja).
- Se entrene offline con historiales clínicos anonimizados.
- Aprenda de nuevos casos clínicos (con feedback implícito del uso real).
- Genere recomendaciones personalizadas sin comprometer privacidad.

Componentes propuestos

1. **Frontend (webapp)**: interfaz del médico durante la consulta.
2. **Sugerencias de diagnóstico**: modelos que generan recomendaciones.
3. **Módulo de resumen clínico**: transforma texto libre en un resumen estructurado.
4. **Pipeline de entrenamiento**: se actualiza con nuevos casos históricos.
5. **Módulo de alertas**: escanea el historial y recomienda acciones preventivas.

Consideraciones

Privacidad vs. conectividad

- ¿Se puede usar la nube si los datos médicos son sensibles?
- ¿Tendría sentido ejecutar parte de la IA en local (on-premise) o incluso offline?

Tiempo real

- ¿Qué componentes deben responder en <300ms?
- ¿El modelo de diagnóstico debe estar siempre disponible localmente?

Consideraciones

Asincronía

- ¿Cómo y cuándo actualizar el modelo?
- ¿Se pueden subir logs anonimizados con retraso?

Comunicación entre módulos

- ¿REST, gRPC, colas de eventos, sincronización por archivos?
- ¿Qué debe ocurrir si un módulo no responde? ¿Fallbacks?