Reis - Fundamentals of Data Engineering. Chapter 3

Joe Reis Matt Housley

November 24, 2025

Capítulo 3. Diseñando una Buena Arquitectura de Datos

Una buena arquitectura de datos proporciona capacidades fluidas a través de cada paso del ciclo de vida de la ingeniería de datos y sus corrientes subyacentes.

¿Qué es la Arquitectura de Datos?

La ingeniería de datos exitosa se construye sobre una arquitectura de datos sólida. Dado que el campo cambia constantemente, las definiciones suelen ser inconsistentes. Para definirla, primero debemos entender el contexto en el que se sitúa: la **Arquitectura Empresarial**.

Arquitectura Empresarial Definida

La arquitectura empresarial tiene muchos subconjuntos, incluidos el negocio, la técnica, la aplicación y los datos.

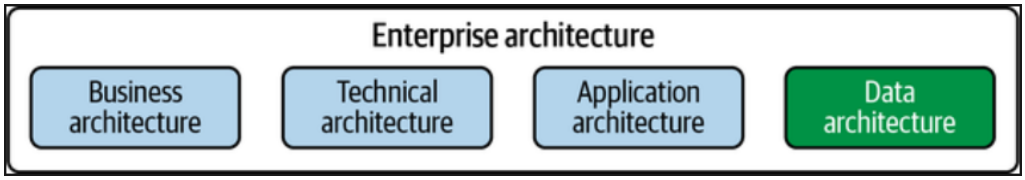

Figura 3-1: La arquitectura de datos es un subconjunto de la arquitectura empresarial

El libro revisa definiciones de líderes de pensamiento:

- **TOGAF:** Abarca toda la información, tecnología, procesos e infraestructura de la empresa o un dominio específico. Cruza múltiples sistemas y grupos funcionales.
- **Gartner:** Disciplina para liderar respuestas proactivas y holísticas a fuerzas

disruptivas, identificando y analizando la ejecución del cambio hacia la visión de negocio deseada.

- **EABOK (Enterprise Architecture Book of Knowledge):** Modelo organizacional que alinea estrategia, operaciones y tecnología.

Definición de los autores:

> La arquitectura empresarial es el diseño de sistemas para **soportar el cambio** en la empresa, logrado mediante **decisiones flexibles y reversibles** alcanzadas a través de una evaluación cuidadosa de las **compensaciones (trade-offs)**.

Puntos clave de esta definición:

1. **Decisiones flexibles y reversibles:**
   - El mundo cambia constantemente; predecir el futuro es imposible.
   - Evitan la "osificación empresarial" (rigidez).
   - **Jeff Bezos y las puertas:**
     - *Puertas de un solo sentido (One-way doors):* Decisiones casi imposibles de revertir (ej. vender una división). Requieren mucha cautela.
     - *Puertas de dos sentidos (Two-way doors):* Decisiones reversibles (ej. elegir una base de datos para un microservicio). Si fallas, puedes volver atrás. Las organizaciones deben tomar estas decisiones rápidamente.
2. **Gestión del cambio:** Las grandes iniciativas deben romperse en cambios más pequeños y reversibles.
3. **Compensaciones (Trade-offs):** Las soluciones técnicas no existen por sí mismas, sino para apoyar objetivos de negocio. Los ingenieros deben gestionar límites físicos (latencia, confiabilidad) y no físicos (costo, complejidad), minimizando la deuda técnica de alto interés.

Arquitectura de Datos Definida

Al igual que la ingeniería de datos es un subconjunto del ciclo de vida de los datos, la arquitectura de datos es un subconjunto de la arquitectura empresarial.

- **TOGAF:** Descripción de la estructura e interacción de los principales tipos y fuentes de datos, activos lógicos y físicos.

- **DAMA DMBOK:** Identificar necesidades de datos y diseñar planos maestros para cumplirlas.

Definición de los autores:

> La arquitectura de datos es el diseño de sistemas para soportar las **necesidades de datos en evolución** de una empresa, logrado mediante decisiones flexibles y reversibles alcanzadas a través de una evaluación cuidadosa de las compensaciones.

Aspectos de la arquitectura de datos:

- **Arquitectura Operacional (El "Qué"):** Requisitos funcionales, personas, procesos. ¿Qué procesos de negocio sirve el dato? ¿Cuál es el requisito de latencia?

- **Arquitectura Técnica (El "Cómo"):** Cómo se ingesta, almacena, transforma y sirve el dato.

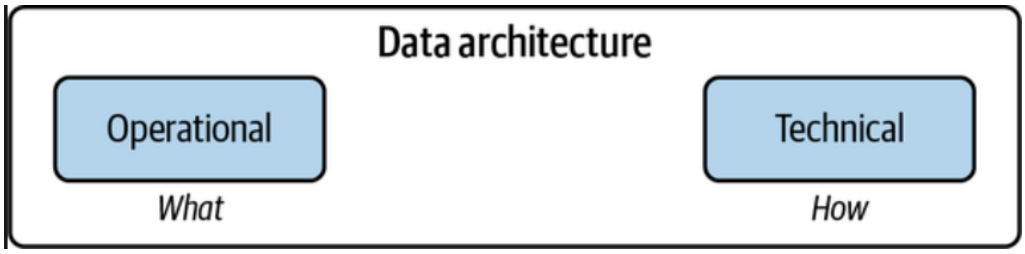

Figura 3-2: Arquitectura de datos operacional y técnica

"Buena" Arquitectura de Datos

> *"Nunca busques la mejor arquitectura, sino la arquitectura menos mala."* — Mark Richards y Neal Ford.

Una buena arquitectura de datos sirve a los requisitos del negocio con un conjunto común y reutilizable de bloques de construcción, manteniendo la flexibilidad.

- **Agilidad:** Es la base. La arquitectura debe evolucionar con el negocio.

- **Mala arquitectura:** Estrechamente acoplada (tightly coupled), rígida, demasiado centralizada o usa herramientas incorrectas.

- Las **corrientes subyacentes** (seguridad, DataOps, orquestación, etc.) forman la base de una buena arquitectura.

Principios de una Buena Arquitectura de Datos

Los autores expanden los pilares de *AWS Well-Architected Framework* y los principios de *Google Cloud* para proponer **9 Principios de la Arquitectura de Ingeniería de Datos**:

Principio 1: Elegir componentes comunes sabiamente

El ingeniero debe seleccionar componentes y prácticas que puedan usarse ampliamente en la organización para facilitar la colaboración y evitar silos.

- Ejemplos: Almacenamiento de objetos, sistemas de control de versiones, orquestación.

- **Equilibrio:** Usar componentes comunes para facilitar la interoperabilidad, pero evitar forzar soluciones "talla única" que obstaculicen la productividad en dominios específicos. Las plataformas en la nube son ideales para adoptar componentes comunes.

Principio 2: Planificar para fallos (Plan for Failure)

"Todo falla, todo el tiempo" (Werner Vogels, AWS). Se deben considerar términos clave:

- **Disponibilidad:** Porcentaje de tiempo que el servicio está operativo.

- **Fiabilidad:** Probabilidad de que el sistema cumpla su función definida durante un intervalo.

- **RTO (Recovery Time Objective):** Tiempo máximo aceptable para una interrupción (¿cuánto tiempo podemos estar caídos?).

- **RPO (Recovery Point Objective):** Pérdida de datos máxima aceptable tras una recuperación (¿cuántos datos podemos perder?).

Principio 3: Arquitectar para la Escalabilidad

- **Scale up (Escalar hacia arriba):** Añadir capacidad para manejar picos (ej. entrenar un modelo con petabytes).

- **Scale down (Escalar hacia abajo):** Reducir capacidad cuando la carga baja para ahorrar costos.

- **Elasticidad:** Capacidad de escalar dinámicamente y automáticamente según la carga.

- **Scale to zero:** Apagarse completamente cuando no está en uso (Serverless).

Principio 4: La Arquitectura es Liderazgo

Los arquitectos deben ser técnicamente competentes pero delegar el trabajo individual.

- No deben ser "cuellos de botella" que toman todas las decisiones (estilo comando y control).

- **Architectus Oryzus (Martin Fowler):** El arquitecto ideal mentora al equipo de desarrollo para que puedan resolver problemas complejos ellos mismos.

Principio 5: Siempre estar arquitectando (Always Be Architecting)

La arquitectura no es un estado estático; es un proceso continuo.

- El arquitecto debe conocer la **arquitectura base** (estado actual), desarrollar una **arquitectura objetivo** (futuro) y un **plan de secuenciación** para llegar allí, ajustándose ágilmente a los cambios.

Principio 6: Construir sistemas débilmente acoplados (Loosely Coupled)

Cuando la arquitectura permite a los equipos probar, desplegar y cambiar sistemas sin depender de otros equipos.

- **Mandato API de Bezos (2002):** Todos los equipos deben exponer datos/funcionalidad a través de interfaces de servicio. No hay enlaces directos ni puertas traseras. Esto permitió a Amazon escalar masivamente (naciendo AWS).

- **Características técnicas:** Sistemas rotos en componentes pequeños, interfaces abstractas (API), cambios internos no afectan a otros, actualizaciones separadas.

- **Características organizacionales:** Equipos pequeños y autónomos que publican detalles abstractos y evolucionan independientemente.

Principio 7: Tomar decisiones reversibles

Apunta siempre a "puertas de dos sentidos". La tecnología cambia rápido; lo que hoy es popular, mañana es obsoleto. Evita la irreversibilidad en el diseño de software.

Principio 8: Priorizar la Seguridad

Responsabilidad compartida y seguridad de confianza cero (*Zero-Trust*).

- **Zero-Trust:** El perímetro endurecido (firewall tradicional) ya no es suficiente. Se asume que las amenazas pueden ser internas y externas.
- **Modelo de Responsabilidad Compartida:** El proveedor de la nube asegura la infraestructura ("seguridad *de* la nube"), el usuario asegura lo que pone dentro ("seguridad *en* la nube").
- Todos los ingenieros de datos son ingenieros de seguridad.

Principio 9: Abrazar FinOps

FinOps es la gestión financiera en la nube.

- Cambio de **CapEx** (Gasto de capital, comprar servidores cada pocos años) a **OpEx** (Gasto operativo, pago por uso).
- Permite escalar pero hace el gasto dinámico. Los ingenieros deben considerar el costo por consulta o procesamiento.
- Cuidado con los "ataques de costo" (ej. descargas excesivas de S3 o bucles infinitos en funciones serverless).

Conceptos Principales de Arquitectura

Dominios y Servicios

- **Dominio:** El área temática del mundo real para la que estás arquitectando (ej. Ventas, Contabilidad).
- **Servicio:** Conjunto de funcionalidades cuyo objetivo es cumplir una tarea (ej. servicio de facturación).
  - Un dominio puede contener múltiples servicios.

– Consejo: Para definir un dominio, habla con los usuarios y stakeholders, no copies ciegamente a otras empresas.

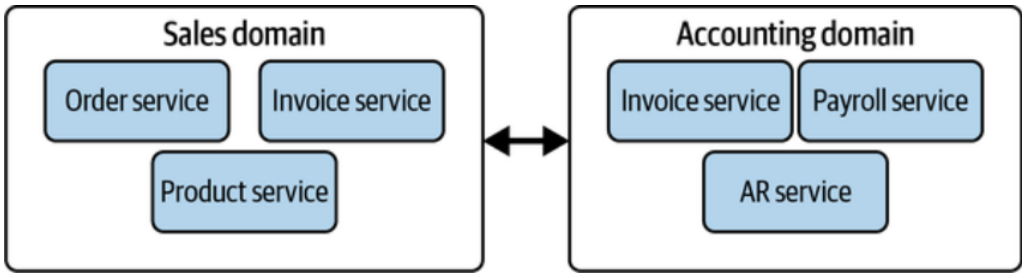

Figura 3-3: Dos dominios (ventas y contabilidad) comparten un servicio común (facturas)

Sistemas Distribuidos

Para lograr escalabilidad y fiabilidad, se usan sistemas distribuidos.

- **Escalado horizontal:** Añadir más máquinas (nodos trabajadores) coordinadas por un nodo líder.

- Provee redundancia (si una máquina muere, otra toma el trabajo).

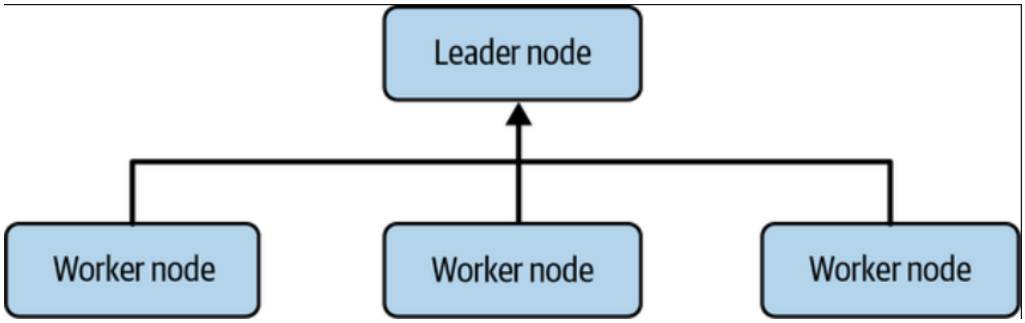

Figura 3-4: Un sistema distribuido horizontal simple utilizando una arquitectura líder-seguidor

Acoplamiento Fuerte vs. Débil (Tight vs. Loose Coupling)

Niveles de Arquitectura (Tiers):

1. **Capa Única (Single Tier):** Base de datos y aplicación en el mismo servidor. Simple pero riesgosa para producción (si falla el servidor, falla todo). No recomendada

para producción.

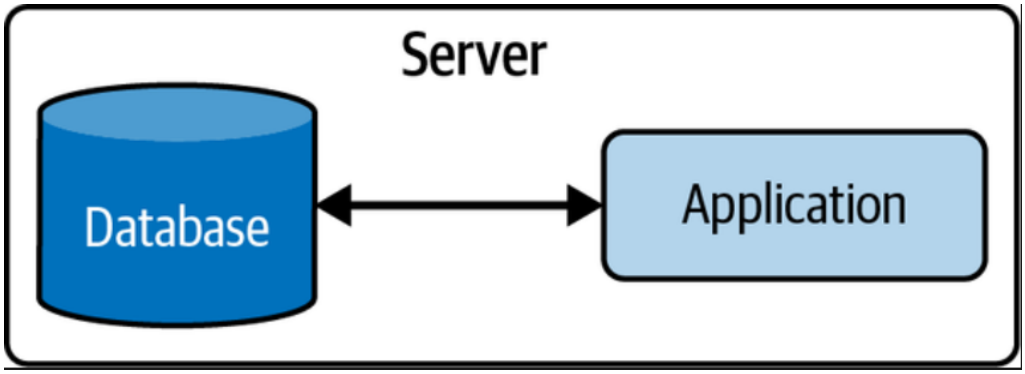

Figura 3-5: Arquitectura de capa única

2. **Multicapa (Multitier):** Separa datos, aplicación y lógica.

    - **Arquitectura de tres capas:** Capa de datos, Capa de aplicación/lógica, Capa de presentación.

    - **Shared-nothing architecture:** Nodos independientes que no comparten memoria ni disco (reduce contención).

    - **Shared-disk architecture:** Nodos comparten disco (útil para fallos de nodos).

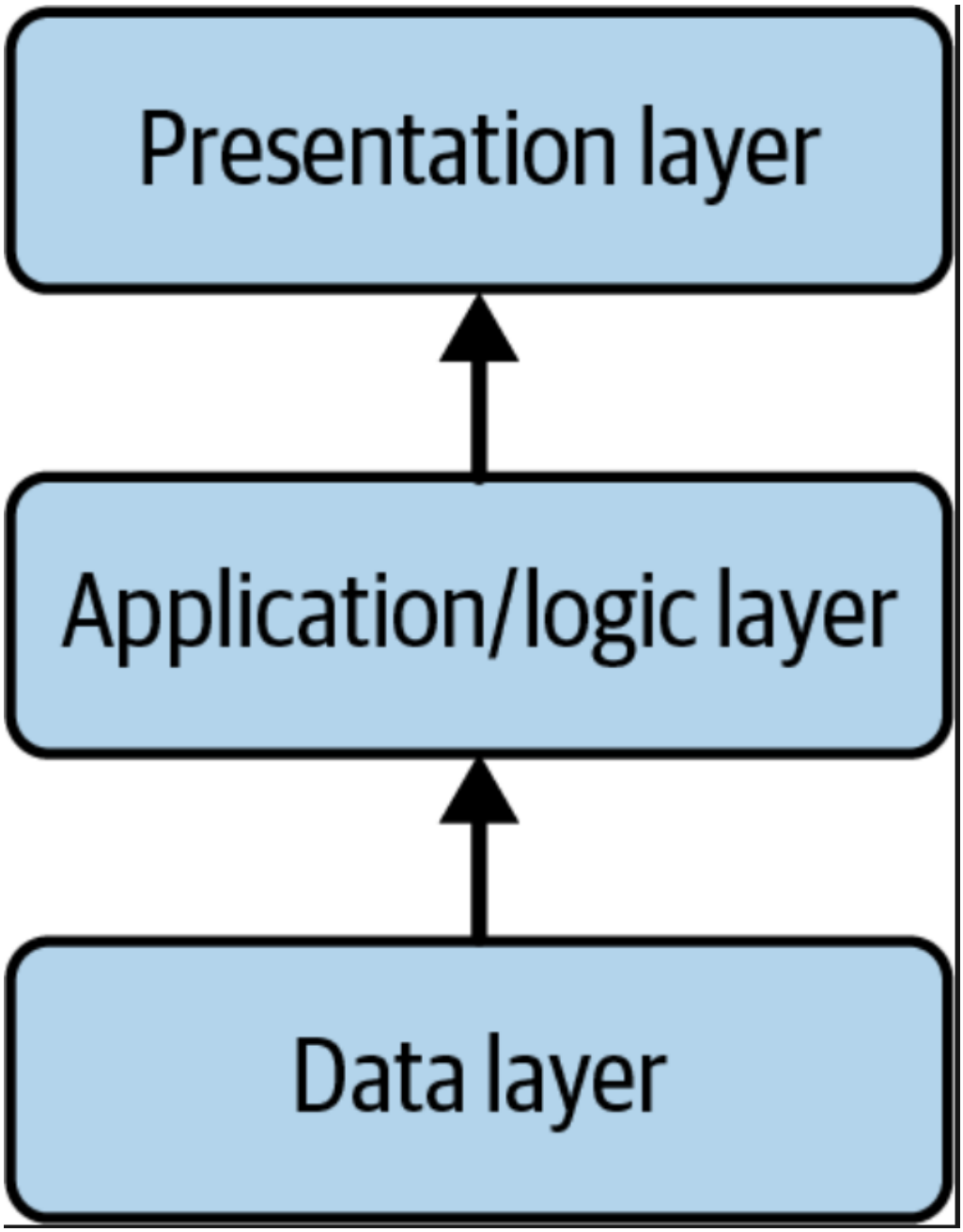

Figura 3-6: Una arquitectura de tres capas

Monolitos:

- Todo bajo un mismo techo (código único, máquina única).
- Acoplamiento técnico y de dominio.
- Difícil de mantener, actualizar o reutilizar componentes.
- A menudo degenera en una "gran bola de lodo" (*big ball of mud*).

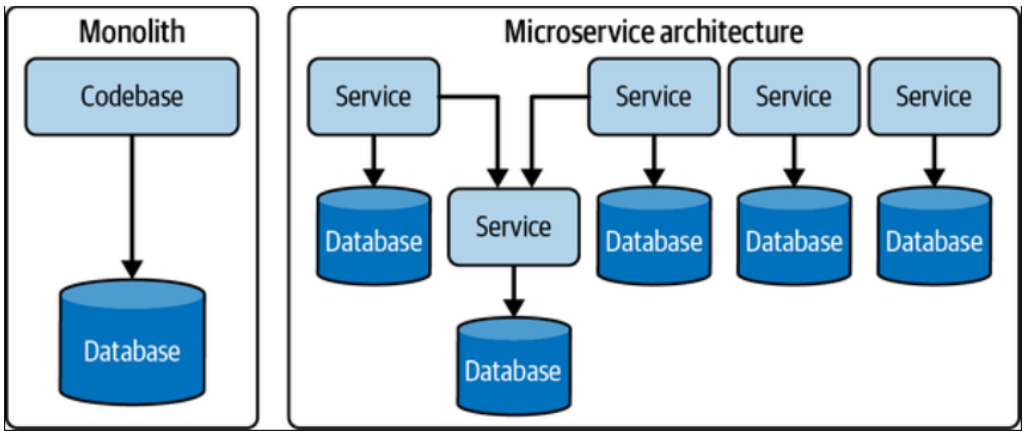

Figura 3-7: Una arquitectura extremadamente monolítica corre toda la funcionalidad dentro de una única base de código

Microservicios:

- Servicios separados, descentralizados y débilmente acoplados.
- Si uno cae, los otros siguen funcionando.
- No siempre es necesario refactorizar un monolito completo; se pueden extraer servicios poco a poco.

Consideraciones para datos: A menudo los Data Warehouses centrales son monolíticos. El enfoque moderno (como Data Mesh) busca descentralizar esto.

- **Consejo:** Usa el acoplamiento débil como ideal pragmático, pero reconoce las limitaciones de las tecnologías.

Acceso de Usuario: Single vs. Multitenant (Multitenencia)

- **Multitenencia:** Compartir sistemas entre múltiples equipos o clientes.
  - Casi todos los servicios en la nube son multitenant.
  - Retos: **Vecinos ruidosos** (un usuario consume todos los recursos y afecta a otros) y **Seguridad** (aislamiento de datos para evitar fugas entre clientes).

Arquitectura Orientada a Eventos (Event-Driven)

El negocio no es estático; ocurren "eventos" (nueva orden, nuevo cliente).

- Workflow: Producción de evento -> Enrutamiento -> Consumo.
- Ventaja: Desacopla servicios. Si un servicio cae, el evento sigue en la cola.

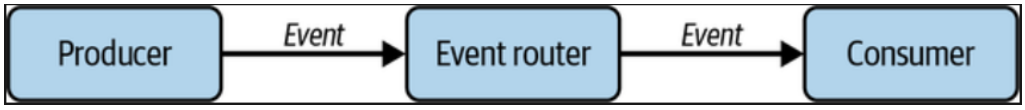

Figura 3-8: En un flujo de trabajo orientado a eventos, un evento es producido, enrutado y luego consumido

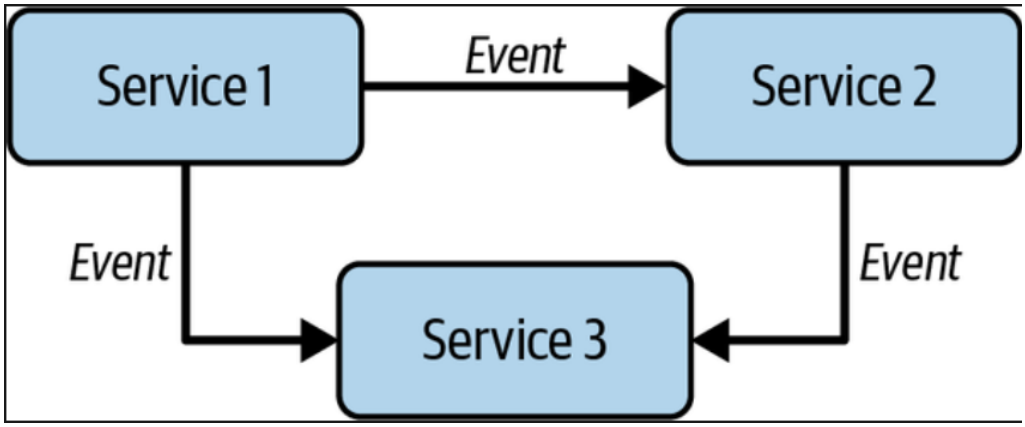

Figura 3-9: En una arquitectura orientada a eventos, los eventos se pasan entre servicios débilmente acoplados

Proyectos Brownfield vs. Greenfield

1. **Brownfield (Campo marrón):** Refactorizar o reorganizar una arquitectura existente (legada).
    - Restringido por decisiones pasadas.
    - Requiere empatía para entender por qué se tomaron esas decisiones.
    - **Estrategia:** Evitar el "Big Bang" (reescritura total). Usar el **Patrón Estrangulador** (*Strangler Pattern*): reemplazar el sistema viejo pieza por pieza incrementalmente.
2. **Greenfield (Campo verde):** Empezar desde cero.
    - Sin restricciones heredadas.
    - **Peligro:** "Síndrome del objeto brillante" o "Desarrollo impulsado por el currículum" (usar tecnología nueva solo porque está de moda, sin justificación de negocio).

Ejemplos y Tipos de Arquitectura de Datos

1. Contexto: por qué Reis habla de "ejemplos y tipos"

Reis dice que la **arquitectura de datos** es algo bastante abstracto, así que conviene **razonar por ejemplos**. En esta sección presentan algunos **patrones de arquitectura muy usados hoy**:

- **Data warehouse**
- **Data lake** (primera generación, "data lake 1.0")
- **Data lakehouse** y **data platform moderna** (convergencia)
- **Lambda architecture**
- **Kappa architecture**
- **Dataflow model / batch como caso especial de streaming**

- Otros ejemplos: **data fabric, data hub, metadata-first, event-driven, live data stack**, etc.

La idea es mostrar cómo cada tipo de arquitectura responde a **necesidades distintas** y tiene **trade-offs** que hay que evaluar.

---

2. Data Warehouse

2.1. Definición general Un **data warehouse** es un **hub centralizado de datos** para **reporting y análisis**. Los datos están **altamente estructurados y modelados** para casos de uso analíticos (OLAP).

Reis cita la definición clásica de Bill Inmon (1989) y la toma como válida: un DW es una colección de datos **orientada a temas, integrada, no volátil y variante en el tiempo**, que soporta la toma de decisiones de management.

2.2. Arquitectura organizacional vs técnica Reis distingue dos niveles:

- **Organizational data warehouse architecture** Cómo se organiza el DW en relación con **estructuras de negocio** (líneas de negocio, áreas, procesos). Ej.: data marts por departamento, modelo corporativo, etc.

- **Technical data warehouse architecture** Cómo se implementa técnicamente:

  - uso de sistemas **MPP** (massively parallel processing),

  - almacenamiento columnar,

  - separación OLTP/OLAP, etc.

Podés tener un DW sin un MPP, o un MPP sin que esté organizado como DW a nivel organizacional.

2.3. ETL, ELT y data marts En la versión clásica:

- Se extraen datos de sistemas de origen (**ETL**: Extract-Transform-Load).

- Se limpian y transforman aplicando reglas de negocio.

- Se cargan en el DW y, muchas veces, en **data marts** específicos por área.

En el enfoque más moderno:

- Se populariza **ELT**: Extract-Load-Transform.
  - Se cargan los datos casi crudos en un **staging** del DW.
  - Se aprovecha la **potencia de cómputo del DW en la nube** para transformar adentro (SQL, herramientas internas).

Los **data marts** se usan para:

- acercar datos a equipos específicos (marketing, finanzas, etc.),
- y agregar más transformación para mejorar el **rendimiento** de las consultas complejas.

2.4. Cloud data warehouse Los **cloud data warehouses** (Redshift, BigQuery, Snowflake, etc.) son una evolución del DW clásico:

- Modelo **pago por uso** (no más contratos millonarios fijos).
- Separación de **compute** y **storage** (objetos en S3/GCS/etc.).
- Capacidad de escalar a **petabytes** y manejar datos estructurados y semiestructurados (JSON, etc.).

Esto hace que el patrón DW sea accesible también a empresas pequeñas.

---

3. Data Lake (data lake 1.0)

3.1. Idea original El **data lake** aparece en la era del "big data" con una idea:

> en vez de imponer mucho esquema, guardemos **todo** (estructurado y no estructurado) en un repositorio central barato y casi ilimitado.

Primera generación ("data lake 1.0"):

- Basada primero en **HDFS** (on-premise) y después en **object storage** en la nube.

- Pensada para almacenar grandes volúmenes y luego montar encima clusters de **MapReduce, Spark, Hive, Presto, etc.** para procesar on-demand.

3.2. Problemas de data lake 1.0 En la práctica, muchos data lakes se convirtieron en:

- **"data swamp"**, **"dark data"**, **WORN** ("write once, read never").
- Repositorios enormes donde:
  - nadie sabe bien qué hay,
  - falta gobernanza y calidad,
  - los costos de operación (clusters, talento especializado) explotan.

Las big tech (Netflix, Facebook, etc.) sí obtuvieron valor porque tenían **equipos enormes y muy especializados**, pero muchas empresas terminaron con un cementerio de datos caro y poco usable.

---

4. Convergencia: next-gen data lakes, data lakehouse y data platform

4.1. Data lakehouse Como reacción a los problemas de los data lakes, aparecen conceptos como el **data lakehouse** (Databricks, etc.):

- Mantiene datos en **object storage** (como un lake).
- Incorpora características típicas de DW:
  - **ACID transactions**,
  - manejo de esquema,
  - control de calidad y gobernanza,
  - estructuras tabulares optimizadas para consultas.
- Permite usar distintos motores (Spark, SQL engines, etc.) sobre los mismos datos.

La idea es **convergencia** entre data warehouse y data lake.

4.2. Cloud DW "lake-like" y data platform Al mismo tiempo, los **cloud DW**:

- Separan compute y storage,
- Aceptan datos semiestructurados,
- Se integran con motores distribuidos (Spark, Beam, etc.).

Reis dice que, en la práctica, la línea entre **DW** y **data lake** se vuelve cada vez más borrosa, y se empieza a hablar de una **data platform** convergente, que combina capacidades de ambos.

4.3. Modern data stack Relacionado con esto, presentan la idea del **modern data stack**:

- Conjunto de componentes **modulares y plug-and-play**, usualmente SaaS/cloud:
  - ingestión/pipelines,
  - almacenamiento,
  - transformación,
  - gobernanza/metadata,
  - monitoreo,
  - visualización y exploración.
- Objetivo: **reducir complejidad** y aumentar **modularidad** y **self-service**, en lugar de tener un único stack monolítico y cerrado.

---

5. Lambda Architecture

5.1. Problema que intenta resolver En los 2010s, con Kafka, Storm, Samza, etc., se volvió popular hacer **streaming / near real-time analytics**. Aparece el problema: ¿cómo combinar **batch** y **streaming** en una sola arquitectura?

La **Lambda architecture** fue una respuesta temprana.

5.2. Estructura En Lambda tenés **tres capas** que operan de forma relativamente independiente:

1. **Batch layer**
    - Procesa grandes volúmenes históricos.
    - Genera vistas agregadas (por ejemplo, en un DW o similar).
2. **Speed (streaming) layer**
    - Procesa eventos en tiempo real con muy baja latencia.
    - Normalmente escribe en alguna store rápida (NoSQL, etc.).
3. **Serving layer**
    - Combina resultados de batch y speed para exponer una visión unificada.

La fuente idealmente es **append-only** (eventos inmutables) y envía datos a ambos caminos (batch y streaming).

5.3. Críticas Reis señala que hoy Lambda:

- No es su primera recomendación para un diseño nuevo.
- Tiende a duplicar lógica (batch vs speed) y complejizar mucho la operación.
- Fue importante históricamente, pero la tecnología y las prácticas avanzaron.

---

6. Kappa Architecture

6.1. Idea central Como reacción a Lambda, Jay Kreps propone la **Kappa architecture**:

En vez de tener un stack separado para batch y otro para streaming, usar un **stream-processing platform** como columna vertebral de toda la arquitectura.

- Todo se modela como un **stream de eventos**.
- La misma infraestructura sirve para:
  - procesar en *tiempo real*,
  - y hacer "batch" **reprocesando** grandes tramos del stream histórico (replay).

6.2. Uso real Reis comenta que:

- No se ve tan adoptada masivamente.
- Razones probables:
  - streaming sigue siendo complejo de operar,
  - es caro a gran escala,
  - batch sigue siendo más eficiente y barato para históricos enormes.

Aun así, Lambda y Kappa sirvieron como **inspiración** para arquitecturas posteriores que intentan unificar batch y streaming.

---

7. Dataflow Model y batch como caso especial de streaming

Otro enfoque más moderno es el **Dataflow model** (Google) y frameworks como **Apache Beam**:

- Todo se ve como **eventos en un stream**.
- Distinción:
  - **unbounded data**: streams "infinitos",

  - **bounded data**: lo que antes llamábamos "batch" (ventana acotada).

- Uso extensivo de **ventanas** (tumbling, sliding, etc.) para agregaciones.

Filosofía:

> "Batch es un caso particular de streaming" (un stream acotado en el tiempo).

Esto permite usar **el mismo modelo y casi el mismo código** para batch y streaming, en lugar de mantener dos caminos completamente distintos como en Lambda.

---

8. Otros ejemplos de arquitecturas de datos

Reis lista brevemente otros patrones que aparecen en la literatura y la industria:

- **Data fabric** Enfoque centrado en integrar datos de múltiples fuentes con fuerte capa de metadata y automatización.

- **Data hub** Nodo central de intercambio/mediación de datos entre sistemas; más orientado a integración que a analítica pura.

- **Scaled architecture** Patrones para escalar data platforms a nivel organización.

- **Metadata-first architecture** Diseñar la plataforma poniendo el énfasis en metadata y gobernanza desde el inicio.

- **Event-driven architecture** Arquitectura centrada en eventos para desacoplar servicios (muy relacionada con streaming).

- **Live data stack** Stack orientado a datos "vivos" / en tiempo (casi) real, que Reis desarrolla más adelante.

No profundizan tanto en cada una, pero la idea es mostrar que el espacio es **muy dinámico** y que continuamente aparecen nuevos nombres y patterns.

---

9. Recapitulando sobre las arquitecturas de datos...

Según **Reis & Housley**, los **ejemplos y tipos de arquitectura de datos** más relevantes incluyen:

- **Data warehouse** (clásico, con ETL/ELT, MPP, data marts, cloud DW).
- **Data lake 1.0** (gran repositorio en HDFS/objetos, pero con problemas de gobernanza y "data swamp").
- **Next-gen data lakes y data lakehouse**, dentro de una **data platform convergente** y un **modern data stack** modular.
- **Lambda architecture** (batch + speed + serving layers para unificar histórico y tiempo real, hoy considerada compleja).
- **Kappa architecture** (usar un único stream-processing backbone para batch y streaming mediante replay).
- El **Dataflow model** y la idea de **batch como caso especial de streaming**.
- Otros patrones: **data fabric, data hub, metadata-first, event-driven, live data stack**, etc.

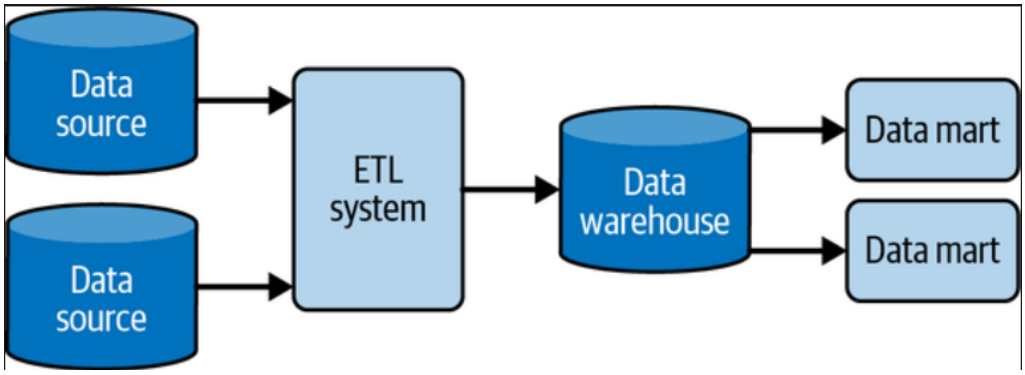

Figura 3-10. *Basic data warehouse with ETL*

- **Qué muestra:**
  Un esquema clásico de *data warehouse* centralizado donde los datos se extraen desde múltiples sistemas fuente, se transforman en una capa de procesamiento intermedio y luego se cargan en el data warehouse (ETL: *Extract–Transform–Load*).
- **Idea clave:**
  Representa la arquitectura tradicional orientada a reporting/BI, con un fuerte énfasis en el modelado previo y en un flujo batch relativamente rígido.

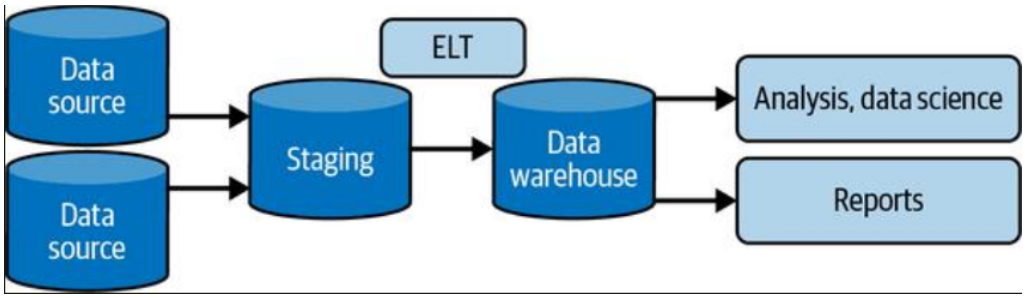

Figura 3-11. *ELT—extract, load, and transform*

- **Qué muestra:**
  Una variante moderna donde los datos se **extraen y se cargan primero** en el data warehouse o en un entorno de almacenamiento escalable, y **las transformaciones se ejecutan después** dentro de ese entorno (ELT: *Extract–Load–Transform*).

- **Idea clave:**
  Ilustra cómo se aprovecha la potencia de cómputo de los data warehouses en la nube para mover lógica de transformación "hacia adentro" de la plataforma, simplificando pipelines y ganando flexibilidad.

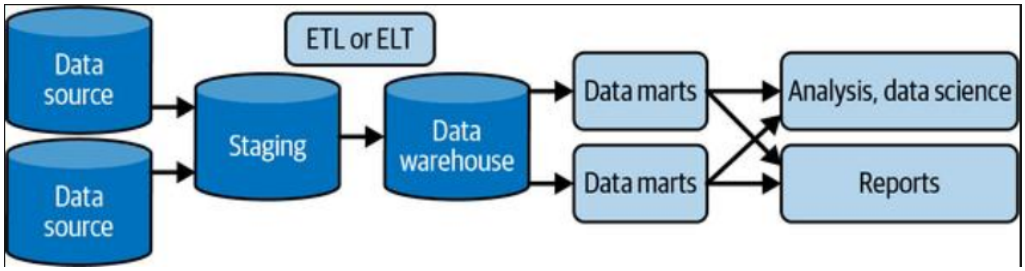

Figura 3-12. *ETL or ELT plus data marts*

- **Qué muestra:**
  Un data warehouse central sobre el cual se construyen **data marts** específicos para distintas áreas de negocio (finanzas, marketing, etc.), alimentados mediante ETL o ELT desde el warehouse.

- **Idea clave:**
  Destaca el patrón de **data warehouse corporativo + data marts departamentales**, que equilibra un modelo común a nivel empresa con vistas optimizadas

para cada dominio.

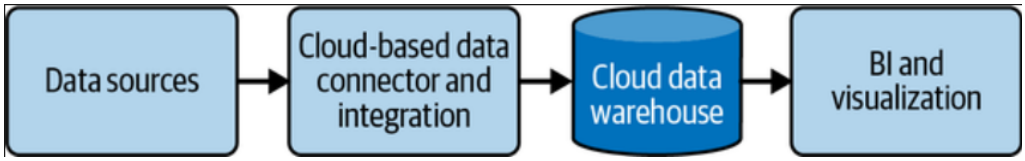

Figura 3-13. *Basic components of the modern data stack*

- **Qué muestra:**
  Los componentes principales de un **modern data stack**: herramientas de ingestión, almacenamiento (warehouse/lake/lakehouse), capa de transformación, catálogo/metadata, herramientas de orquestación/observabilidad y capas de consumo (BI, notebooks, ML, etc.).

- **Idea clave:**
  Resume la visión modular y *SaaS-first* de las plataformas de datos modernas, donde cada función (ingestión, transformación, serving, gobernanza) se resuelve con componentes especializados y acoplados de forma laxa.

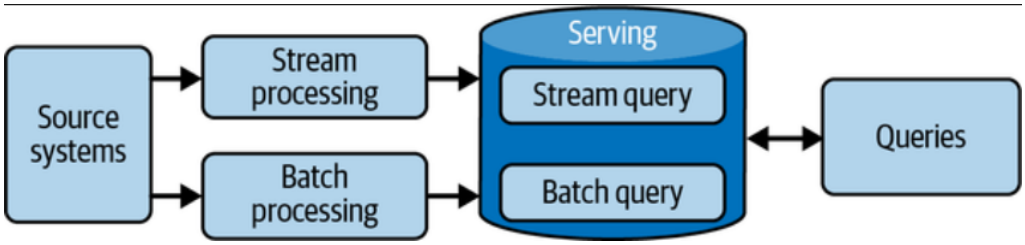

Figura 3-14. *Lambda architecture*

- **Qué muestra:**
  La **Lambda architecture** con sus tres capas:

  – *Batch layer* (procesa datos históricos y genera vistas batch),

  – *Speed layer* (procesa eventos recientes en tiempo real),

– *Serving layer* (combina resultados de batch y speed).

- **Idea clave:**
  Ejemplifica un patrón que intenta unificar histórico + tiempo real, pero que en la práctica introduce **duplicación de lógica y complejidad operativa**.

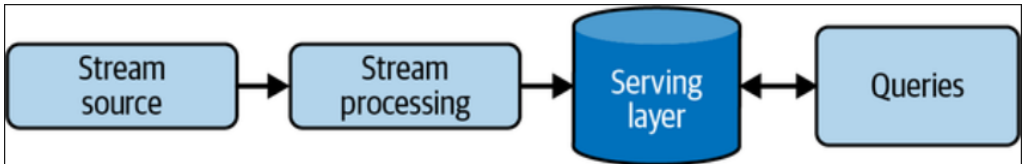

Figura 3-15. *Kappa architecture*

- **Qué muestra:**
  La **Kappa architecture**, en la que todos los datos se tratan como un **stream de eventos**.
  La misma infraestructura de procesamiento de streams se utiliza tanto para el *near real-time* como para reprocesar históricos (replay) en lugar de tener caminos separados batch/stream.

- **Idea clave:**
  Plantea un enfoque "stream-first": **un solo pipeline basado en eventos** para cubrir casos batch y streaming, reduciendo duplicación conceptual respecto de Lambda.

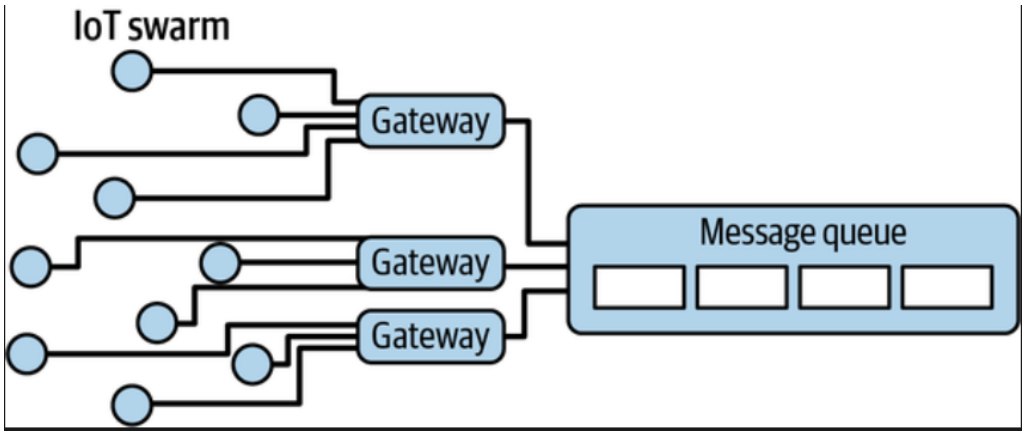

Figura 3-16. *A device swarm (circles), IoT gateways, and message queue with messages*

- **Qué muestra:**
  Un **enjambre de dispositivos IoT** (sensores, actuadores) que se conectan a **gateways IoT**, los cuales a su vez publican datos en una **cola de mensajes** (message queue).

- **Idea clave:**
  Representa el patrón de **ingestión IoT**: los dispositivos no se conectan directamente a la plataforma de datos, sino que se agrupan detrás de gateways que gestionan conectividad, agregación, buffering y envío confiable hacia el backend.

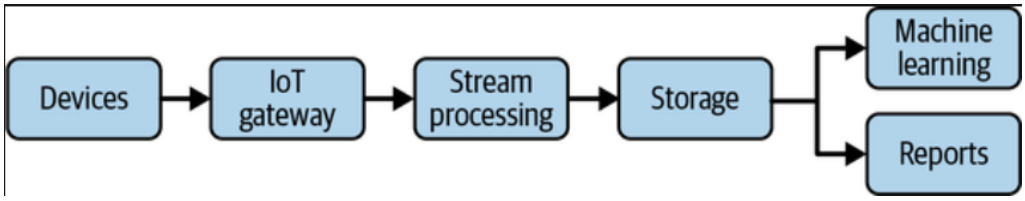

Figura 3-17. *IoT serving pattern for downstream use cases*

- **Qué muestra:**
  Cómo los datos IoT, una vez ingeridos y procesados, se distribuyen hacia distintos **casos de uso downstream**:

  - analítica y dashboards,
  - modelos de ML,
  - sistemas de alerta,
  - aplicaciones que reconfiguran dispositivos, etc.

- **Idea clave:**
  Ilustra el **patrón de serving para IoT**: la plataforma de datos actúa como intermediario entre el mundo físico (dispositivos) y los consumidores analíticos/operacionales, habilitando *feedback loops* (p. ej. enviar nuevas configuraciones a los dispositivos).

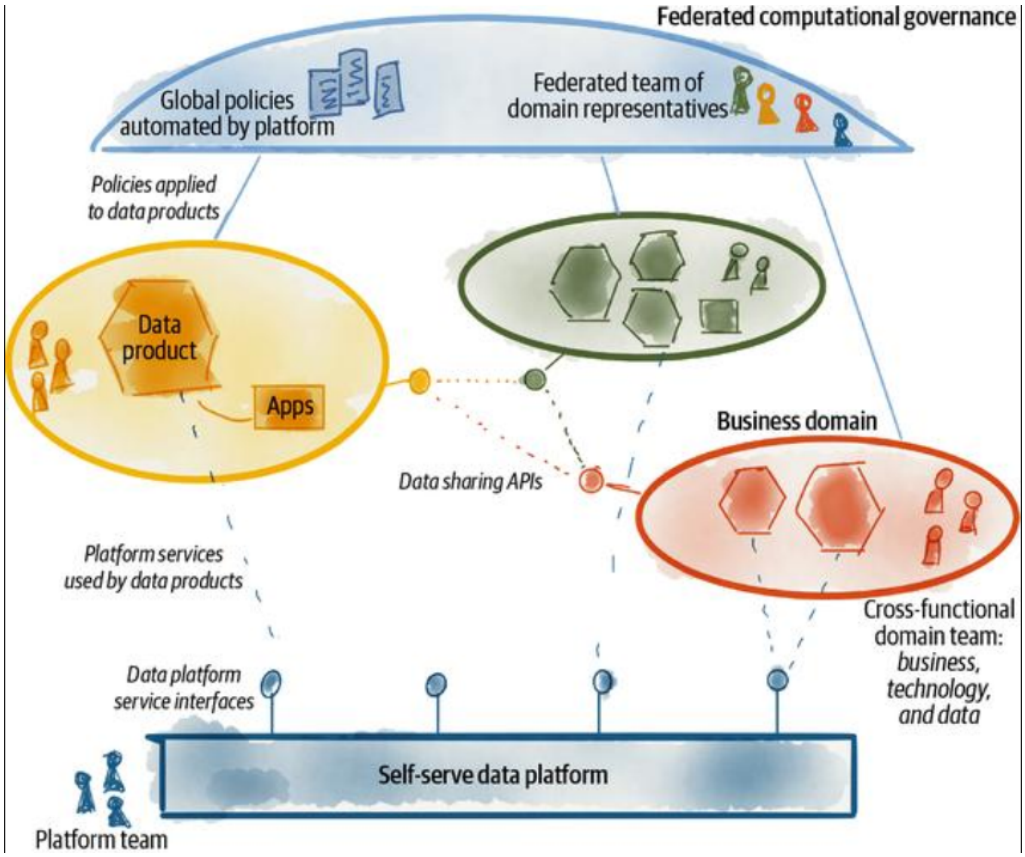

Figura 3-18. *Simplified example of a data mesh architecture*

- **Qué muestra:**
  Un **ejemplo simplificado de arquitectura de data mesh**, con varios dominios de negocio que exponen **data products** propios, apoyados por una **plataforma de datos self-service** y una **gobernanza federada**.

- **Idea clave:**
  Conecta el capítulo con las ideas de *data mesh* (Zhamak Dehghani):

  – datos organizados por dominios,

  – cada dominio responsable de la calidad y publicación de sus productos de datos,

  – una plataforma común que estandariza herramientas, contratos y gobernanza.

¿Quién está involucrado en el diseño?

La arquitectura de datos ya no se diseña en una torre de marfil.

Cuando leas "ya no se diseña en una torre de marfil", significa que la arquitectura de datos moderna:

- Es Pragmática, no Teórica: Se basa en lo que funciona en la realidad, no en ideales académicos.
- Es Colaborativa: No es dictada por una sola persona "sabia" desde arriba, sino construida con el feedback de quienes van a usar y mantener el sistema.
- Requiere "Manos en la masa": Los arquitectos modernos deben entender el código y la ingeniería, no solo hacer dibujos.

- Los ingenieros de datos deben trabajar junto a arquitectos dedicados (o asumir ese rol en empresas pequeñas).
- Deben trabajar con stakeholders del negocio para evaluar compensaciones (trade-offs).
- La distinción entre "arquitectura" e "ingeniería" está desapareciendo; se vuelve más ágil.

Conclusión

Has aprendido cómo la arquitectura encaja en el ciclo de vida, qué define una "buena" arquitectura y los patrones principales (Warehouse, Lake, Lakehouse, Modern Stack, Lambda/Kappa, IoT, Data Mesh). La clave es estudiar profundamente las compensaciones (trade-offs) de cada una para tomar decisiones concretas y valiosas.