Facsímil 08

La ciencia de los datos

Capítulo 01

Datos, datasets y linaje: la primera decisión de IA

686f6c61 · 2026

Capítulo 01: Datos, datasets y linaje: la primera decisión de IA

Entrando en el tema

Este facsímil cambia el foco. Hasta ahora hemos hablado de modelos, APIs, agentes, operación, evaluación, calibración e interpretabilidad. Pero todos esos sistemas tienen una pieza anterior que condiciona casi todo: **qué datos entran, de dónde vienen, para qué pueden usarse y cómo sabemos que no nos estamos engañando**.

La frase central del capítulo es esta:

> Un dataset no es una carpeta con filas. Es una decisión técnica congelada.

Qué deberías poder hacer al terminar

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Distinguir dato, ejemplo, dataset y artefacto derivado.	No llamas “datos” a cualquier texto metido en un prompt.
Diseñar un contrato de datos mínimo.	Defines columnas, splits, licencias, sensibilidad, owner y checks.
Explicar linaje.	Puedes responder qué fuente produjo cada fila y qué hash identifica el snapshot.
Detectar leakage básico.	Sabes buscar duplicados o equivalencias entre train, validation y test.
Separar uso de entrenamiento, evaluación y consulta.	No usas el mismo dato para todo si su licencia o finalidad no lo permite.
Conectar datos con RAG, fine-tuning y evals.	Ves que documentos, chunks, embeddings, labels y trazas son artefactos de datos.
Ejecutar un gate de datos.	Generas un reporte, una dataset card y una decisión técnica reproducible.

Estos resultados no son una lista de tareas sueltas. Forman una cadena: primero sabes qué dato tienes, después decides para qué puede usarse, luego congelas una versión y por último dejas una evidencia que otra persona pueda revisar. Esa cadena evita una trampa muy común en IA: pensar que el dato es un insumo pasivo cuando en realidad define qué puede aprender el sistema, qué puede evaluar y qué riesgos puede esconder.

La ciencia de datos en IA no empieza entrenando. Empieza preguntando: **¿puedo usar este dato para esta decisión?**

La escena: el modelo falla, pero el bug estaba en el dataset

Imagina un asistente académico que responde dudas sobre matrícula, becas, pagos y horarios. El equipo detecta errores en producción y lo primero que se propone es cambiar de modelo. Parece razonable: si la respuesta falla, el modelo será peor de lo esperado.

Pero al mirar los datos aparecen cosas menos vistosas:

HALLAZGO	CONSECUENCIA
Varias respuestas de evaluación también estaban en entrenamiento.	La métrica era demasiado optimista.
Algunos documentos del corpus estaban obsoletos.	El RAG citaba bien, pero citaba una fuente vieja.
Una licencia permitía consulta interna, no entrenamiento.	El fine-tuning propuesto no era aceptable.
El campo `producto` cambió de nombre en producción.	El modelo recibía una feature rota.
Las etiquetas se pusieron con criterios distintos según el equipo.	La métrica mezclaba desacuerdos humanos con errores del modelo.

En ese punto, cambiar de modelo es secundario. El sistema no necesita más brillo; necesita DataOps.

Qué no es un dataset

Un dataset no es “unos CSV que encontré”, una carpeta de PDFs, una tabla de tickets o un export rápido de una herramienta interna. Eso puede ser una fuente de datos, pero todavía no es un dataset listo para ingeniería de IA.

Tampoco es un objeto neutral. La forma de recogerlo, filtrar filas, etiquetar casos, borrar columnas, crear splits y elegir qué queda fuera define qué aprenderá o evaluará el sistema.

Y no es intercambiable entre usos. Un dato puede servir para consultar en RAG, pero no para entrenar. Puede servir para evaluación interna, pero no para publicar resultados. Puede servir con metadatos, pero no si se separa de su fuente.

CONFUSIÓN	LECTURA DE INGENIERÍA
“Tengo datos, ya puedo entrenar”.	Primero contrato, linaje, licencia, sensibilidad y splits.
“Es texto público, puedo usarlo para todo”.	Hay que revisar licencia, finalidad, atribución y restricciones.
“Los embeddings no son texto”.	Son datos derivados y deben protegerse como parte del corpus.
“La eval salió alta”.	Si hay leakage, duplicados o etiquetas débiles, la métrica no vale lo que parece.
“Lo arreglamos limpiando después”.	La limpieza sin contrato crea versiones imposibles de reproducir.

Datasheets for Datasets propuso documentar motivación, composición, recogida, preprocesamiento, usos y distribución de datasets para mejorar transparencia y responsabilidad técnica.[1] Data Cards sigue esa línea con fichas de dataset orientadas a decisiones de uso y documentación práctica.[2]

Qué sí es un dataset para IA

En este facsímil llamaremos dataset a un conjunto de ejemplos con datos, metadatos, finalidad, propietario, versión, permisos, splits y checks verificables.

Esto no es una ley matemática, es un contrato operativo. Un dataset profesional debería dejar visibles estas piezas:

PIEZA	SIGNIFICADO	EJEMPLO
Entradas o features	Lo que verá el sistema.	Texto del ticket, producto, canal, idioma.
Etiquetas o referencias	Lo que se aprende, compara o evalúa.	`answer`, `ask_more`, `escalate`.
Metadatos	Información que permite decidir si el dato es usable.	Fuente, fecha, owner, licencia, sensibilidad.
Splits	Separación de usos.	`train`, `validation`, `test`.
Linaje	Camino que explica origen y transformaciones.	Hash del snapshot, documento origen, versión de chunking.
Versión	Identificador estable del snapshot.	`support-cases@2026-06-06`.
Checks de calidad	Reglas ejecutables que aceptan, revisan o bloquean.	Schema, duplicados, missing, licencias, leakage.

Cada fila también debe poder leerse como un objeto trazable. En la práctica eso significa que no basta con tener `text` y `label`: necesitamos `case_id`, `source_id`, `split`, `created_at`, `license`, `owner` y `consent_scope`. Si mañana una respuesta falla, esas columnas permiten reconstruir de dónde salió el ejemplo, qué permiso tenía, qué split ocupaba y qué contrato aprobó su uso.

La diferencia con “un CSV” es brutal: ahora el dataset tiene contrato. Puedes auditarlo, versionarlo, discutirlo y bloquear su uso si no cumple.

Cómo se estructura un dataset

Un dataset se estructura según la tarea. Esta frase parece obvia, pero evita muchos errores. No tiene la misma forma un dataset para clasificar tickets, un corpus RAG, un conjunto de preferencias, una tabla de features o una traza de agente.

La pregunta que conviene hacerse primero no es “qué formato uso”, sino **cuál es la unidad de decisión**. En clasificación puede ser una fila. En RAG puede ser un chunk con su documento origen. En preferencias puede ser una comparación entre dos respuestas. En agentes puede ser una trayectoria completa. Si no defines esa unidad, acabas mezclando cosas que parecen datos parecidos, pero que responden a decisiones distintas.

También hay una pregunta de tiempo: ¿este dato describe algo que ya estaba disponible cuando el sistema debía decidir, o se calculó después? Esta distinción marca la frontera entre un dataset honesto y un dataset que da ventaja irreal al modelo. En ingeniería de IA, el

tiempo del dato importa tanto como el valor del dato.

Dataset tabular

Un dataset tabular suele tener una fila por ejemplo y columnas para entrada, etiqueta y metadatos:

```
case_id | split | product   | label
d001    | train | matricula | answer
```

COLUMNA	POR QUÉ IMPORTA
`case_id`	Permite rastrear una fila concreta.
`split`	Evita mezclar entrenamiento y evaluación.
`source_id`	Conecta el ejemplo con su fuente original.
`label`	Define lo que aprende o mide el sistema.
`license`	Decide si puede entrenarse, evaluarse o solo consultarse.
`pii_risk`	Define controles de tratamiento y revisión.

Este formato sirve para clasificación, regresión, scoring, routing, análisis por segmentos y evaluación con referencias.

La lectura importante es que una fila no debería ser solo “texto y etiqueta”. Si `source_id` falta, no sabes de dónde salió el caso. Si `split` falta, no sabes si el ejemplo mide o entrena. Si `license` falta, no sabes si el uso que quieres hacer es aceptable. Y si `pii_risk` falta, el sistema puede tratar todos los casos igual aunque unos necesiten más cuidado que otros.

En un proyecto real, muchas discusiones se resuelven mirando estas columnas. Producto pregunta si el asistente puede contestar consultas de becas; datos mira cuántos ejemplos de `becas` hay por split; legal o compliance mira permisos; ingeniería mira si el schema se mantiene estable. Esa es la diferencia entre un CSV suelto y un dataset gobernable.

Dataset JSONL para LLMs

En LLMs se usa mucho JSONL porque cada línea puede representar una conversación, una instrucción, una preferencia o una traza:

```
{"prompt":"Resume","response":"Falta dato","split":"train"}
{"prompt":"Decide","response":{"action":"ask_more"},"split":"test"}
```

Lo importante no es que sea JSON. Lo importante es que cada línea sea independiente, versionable y validable. Si una línea trae `messages`, `tools`, `expected_output`, `rubric_id` o `trace_id`, el contrato debe declararlo.

JSONL se usa mucho porque encaja bien con pipelines: puedes leer línea a línea, validar ejemplos de forma aislada, partir en splits y detectar qué registro falló sin cargar un archivo gigante completo. Para un alumno de ingeniería, esta ventaja es muy concreta: si la línea 842 rompe el contrato, no tienes que interpretar una conversación entera o un export opaco; puedes señalar el ejemplo exacto.

Hay otra consecuencia: el orden de campos no es el contrato. El contrato vive en lo que esperas de cada campo. `prompt` puede ser texto plano, pero `response` puede ser texto, JSON estructurado o una lista de mensajes. Si mañana cambias de `prompt` / `response` a `messages`, no has cambiado solo nombres: has cambiado la interfaz de datos que alimenta al sistema.

Dataset de RAG

Un corpus RAG no es solo texto. Cada chunk debería traer metadatos:

```
{
  "document_id": "doc-pay-001",
  "chunk_id": "doc-pay-001#001",
  "source_uri": "intranet://academico/pagos/justificantes",
  "version": "2026-05-02",
  "text": "El justificante de pago se revisa...",
  "license": "internal_retrieval_allowed",
  "expires_at": "2026-12-31"
}
```

CAMPO	PREGUNTA QUE RESPONDE
`document_id`	¿Qué documento produjo este chunk?
`chunk_id`	¿Qué fragmento exacto se recuperó?
`source_uri`	¿Dónde está la fuente original?
`version`	¿Qué versión estaba vigente?
`expires_at`	¿Cuándo caduca esta evidencia?
`license`	¿Puede recuperarse, indexarse o mostrarse?

Si un RAG falla, muchas veces el problema no es el modelo. Puede ser un chunk sin fecha, una fuente obsoleta, un embedding generado con otro modelo, metadatos incompletos o un índice que no refleja el snapshot esperado.

Piensa en una consulta académica sobre pagos. Si el retriever encuentra un fragmento correcto pero de una normativa antigua, el modelo puede sonar seguro y aun así guiar mal. Por eso `version` y `expires_at` no son decoración: permiten decidir si una evidencia sigue viva. En RAG, recuperar “algo parecido” no basta; hay que recuperar algo pertinente, vigente y permitido.

Además, el chunk es un dato derivado. Se ha creado al partir un documento, quizá después de limpiar HTML, pasar OCR, eliminar cabeceras y generar embeddings. Cada paso puede cambiar el resultado. Si no guardas el `document_id`, la versión del chunking y el modelo de embedding, reconstruir un fallo se vuelve una conversación de memoria en vez de una investigación técnica.

Dataset de preferencias

Un dataset de preferencias enseña comparaciones:

```
{
  "prompt": "Responde una consulta sin evidencia",
  "chosen": "No tengo evidencia suficiente...",
  "rejected": "La anulacion siempre se puede hacer...",
  "preference_policy": "abstencion_con_evidencia"
}
```

Aquí el dato no dice solo “esta respuesta es buena”. Dice: “entre estas dos respuestas, bajo esta política, preferimos esta”. Eso exige documentar quién etiquetó, con qué rúbrica, en qué fecha y para qué entrenamiento o evaluación se puede usar.

La parte delicada es que una preferencia siempre depende de criterio. En un asistente académico quizá preferimos una respuesta que pregunta por documentación antes que otra que inventa una solución. En un asistente de programación quizá preferimos una respuesta que añade test y explica el riesgo. La preferencia no flota en el aire: necesita una política escrita.

Si no guardas `preference_policy`, un equipo puede interpretar que `chosen` significa “más amable”, otro que significa “más corta” y otro que significa “más correcta”. El dataset parecerá grande, pero estará enseñando señales mezcladas. Para post-entrenamiento y evaluación comparativa, esa ambigüedad es veneno lento: no rompe el script, pero degrada la decisión.

Dataset de trazas

En agentes, un dataset útil puede ser una traza:

```
{
  "trace_id": "tr_001",
  "goal": "resolver ticket",
  "state": "falta evidencia",
  "action": "retrieve_documents",
  "observation": "2 chunks recuperados",
  "outcome": "ask_more"
}
```

Este formato sirve para evaluar trayectorias, herramientas, costes, pasos innecesarios, criterios de parada y handoffs. Si solo guardamos la respuesta final, perdemos el dato que explica cómo llegó el sistema hasta allí.

La traza cambia la pregunta. Ya no miras solo si la respuesta final fue aceptable; miras si el sistema eligió bien los pasos. ¿Consultó una herramienta cuando hacía falta? ¿Pidió aclaración cuando la evidencia era insuficiente? ¿Repitió una llamada sin ganar información? ¿Terminó por el motivo correcto? Esa información es oro para ingeniería porque permite mejorar el sistema sin tocar necesariamente el modelo.

También convierte observabilidad en dataset. Las trazas que vimos en facsímiles anteriores no sirven solo para depurar una incidencia puntual. Bien muestreadas y revisadas, pueden transformarse en evals, casos de regresión, datasets de herramientas o ejemplos para entrenar políticas de orquestación.

Dataset de features

En ML clásico y sistemas híbridos aparece otra estructura: una entidad con features calculadas en un tiempo concreto.

```
entity_id | event_time | wait_days | label
s001      | 2026-05-01 | 12        | urgent
```

Aquí hay una trampa crítica: las features deben existir en el momento de la predicción. Si calculas una feature usando información posterior al evento, el modelo aprende con ventaja irreal. Este error se llama leakage temporal y suele producir métricas preciosas en notebook y decepción en producción.

El campo `event_time` es el ancla. Si quieres predecir el 1 de mayo si un caso será urgente, no puedes usar una variable calculada el 5 de mayo. Parece una obviedad, pero ocurre mucho cuando se agregan tablas históricas sin respetar la fecha de disponibilidad de cada columna.

Por eso un feature store no es simplemente “una base de datos con variables”. Su valor está en mantener la misma definición de feature para entrenamiento e inferencia, controlar frescura, guardar versiones y evitar que el notebook use una realidad distinta a la que verá el servicio en producción.

Taxonomía práctica de datasets en IA

TIPO DE DATASET	UNIDAD MÍNIMA	USO TÍPICO	RIESGO PRINCIPAL
Tabular supervisado	Fila con features y etiqueta.	Clasificación, regresión, routing.	Leakage, desbalance, etiqueta ruidosa.
Corpus RAG	Documento o chunk con metadatos.	Recuperación y citas.	Fuente obsoleta, chunk sin linaje.
Instrucciones	Entrada y salida esperada.	SFT, evaluación, formato.	Enseñar estilo sin cubrir casos reales.
Preferencias	Prompt, respuesta elegida y rechazada.	Preferencias, ranking, post-entrenamiento.	Rúbrica ambigua o etiqueta inconsistente.
Trazas	Estado, acción, observación y resultado.	Agentes y EvalOps.	Guardar solo el final y perder diagnóstico.
Features	Entidad, tiempo y variables calculadas.	Modelos online/offline.	Desalineación entre entrenamiento e inferencia.
Multimodal	Texto, imagen/audio/video y metadatos.	Visión, documentos, OCR, audio.	Calidad de OCR, resolución, permisos.
Producción muestreada	Run real revisada.	Drift, regresiones, mejora continua.	Sesgo de muestreo y falta de owner.

Esta tabla no pretende encerrar todos los casos. Sirve para que el alumno aprenda a hacer una separación mental: **qué unidad estoy midiendo, qué uso tendrá y cuál es el riesgo dominante**. Si el dataset es RAG, el riesgo dominante suele estar en la fuente y el linaje. Si es preferencias, en la rúbrica. Si es features, en el tiempo y la consistencia entre entrenamiento e inferencia.

La decisión práctica es más concreta de lo que parece. Si el dataset es de instrucciones, el equipo debe mirar diversidad de tareas, estilo esperado y cobertura de casos límite. Si es de preferencias, debe mirar quién eligió la respuesta ganadora, con qué criterio y qué ocurre cuando dos respuestas son casi igual de buenas. Si es de producción muestreada, debe mirar cómo se eligieron las runs revisadas: no es lo mismo revisar solo quejas que revisar una muestra estratificada por canal, idioma y producto. Cada tipo de dataset trae una forma distinta de engañarse.

Una buena práctica es nombrar el dataset por su uso, no solo por su origen. `tickets_soporte.csv` dice poco. `support_cases_eval_v2026_06` ya indica una finalidad. `support_cases_sft_v2026_06` indica otra. Puede venir de la misma fuente, pero no tiene el

mismo contrato ni debería tener los mismos permisos.

Hugging Face Datasets popularizó una interfaz donde un dataset puede cargarse como `DatasetDict`, con splits como `train`, `validation` y `test`, y operaciones reproducibles de carga, transformación y procesamiento.[3] No hace falta usar esa librería para aprender la idea: un dataset moderno debe declarar estructura, particiones y transformaciones de forma explícita.

Fecha de corte del estado del arte

Fecha de corte: 6 de junio de 2026.
Fuentes consultadas: Datasheets for Datasets, Data Cards, Model Cards, Hidden Technical Debt in Machine Learning Systems, Software Engineering for Machine Learning, TFX, ML Test Score, ODCS, TensorFlow Data Validation, ML Metadata, OpenLineage, DataHub, Great Expectations, Feast, Evidently, DVC, lakeFS, Delta Lake, Apache Iceberg, Apache Hudi, Apache Parquet, Apache Arrow, The Dataflow Model, Apache Airflow, Dagster, dbt y Hugging Face Datasets.

Lo estable es la disciplina: datos versionados, schema verificable, linaje, documentación, validación, separación de splits, evaluación por segmentos y monitorización de cambios. Lo coyuntural son herramientas, paneles, proveedores y formatos exactos.

Las fuentes de este bloque no dicen todas lo mismo, y eso es bueno. Unas hablan de documentación, otras de pipelines, otras de contratos, otras de linaje y otras de monitorización. Juntas dibujan una idea muy importante: en IA, la calidad del sistema no se puede separar de la vida del dato. El dato nace, cambia, se transforma, se versiona, se consume y se degrada.

Sculley y otros explicaron que los sistemas de ML acumulan deuda técnica especial: dependencias de datos, realimentaciones escondidas, configuraciones frágiles y cambios no locales.[4] Amershi y otros mostraron desde Microsoft que construir ML exige prácticas de ingeniería diferentes al software clásico, especialmente alrededor de datos, experimentos y evolución del sistema.[5]

TFX formalizó una plataforma de ML a escala de producción donde ingestión, validación, transformación, entrenamiento, evaluación y serving forman un pipeline reproducible.[6] El ML Test Score propuso una rúbrica para madurez de producción que incluye tests de datos, validación, monitorización y gestión de cambios.[7]

En herramientas actuales, ODCS define una estructura abierta para contratos de datos con secciones de fundamentos, schema, referencias, calidad, soporte, equipo, roles y acuerdos de servicio.[8] TensorFlow Data Validation analiza datos, genera estadísticas, infiere schema y detecta anomalías contra expectativas.[9] ML Metadata registra artefactos, ejecuciones y contextos para recuperar linaje en workflows de ML.[10]

OpenLineage aporta un estándar abierto para capturar metadatos de linaje en ejecuciones de jobs.[11] DataHub se presenta como catálogo de datos para metadatos, búsqueda, linaje, perfilado, gobernanza y contratos.[12] Great Expectations organiza calidad de datos mediante expectations reutilizables.[13]

Feast es un feature store que separa offline store para generar datos de entrenamiento y online store para servir features de baja latencia.[14] Evidently documenta métricas de drift para comparar distribuciones y detectar cambios en datos o modelos.[15] DVC, lakeFS, Delta Lake y Apache Iceberg tratan el problema de reproducibilidad y versionado desde ángulos distintos: proyectos ML versionados, data lakes con semántica tipo Git, time travel, schema enforcement, snapshots y evolución de schema.[16][17][18][19]

La lección práctica para este facsímil: **los datos no son un paso previo; son parte del sistema**.

Arquitectura de datos para IA

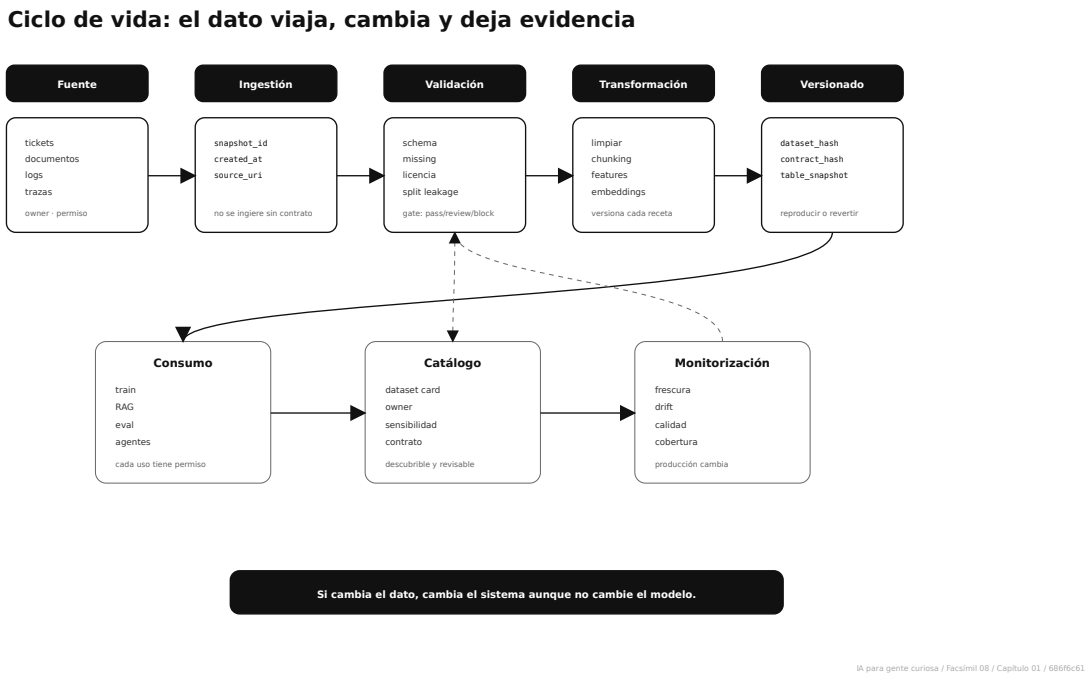
Una arquitectura mínima de datos para IA tiene varias capas. No todas hacen falta en un proyecto pequeño, pero todas deben existir como pregunta mental:

CAPA	QUÉ GUARDA	PREGUNTA QUE RESPONDE
Fuente	Tabla, documento, log, traza, imagen o evento original.	¿De dónde salió este dato?
Ingestión	Copia controlada del dato.	¿Cuándo entró y bajo qué contrato?
Validación	Estadísticas, schema, expectations y anomalías.	¿Cumple lo prometido?
Transformación	Limpieza, chunking, OCR, features, embeddings.	¿Qué se cambió y con qué versión?
Versionado	Snapshot, hash, commit, tabla Delta/Iceberg, DVC/lakeFS.	¿Puedo reconstruir el mismo dataset?
Catálogo	Owner, descripción, sensibilidad, contrato, documentación.	¿Quién responde por esto y para qué sirve?
Linaje	Jobs, inputs, outputs, columnas y artefactos derivados.	¿Qué rompemos si cambia esta fuente?
Consumo	Entrenamiento, RAG, eval, dashboard, agente, producto.	¿Qué sistema usa este dato?
Monitorización	Drift, frescura, calidad, cobertura, feedback.	¿Producción sigue pareciéndose a lo validado?

La arquitectura correcta depende del tamaño. Para un proyecto didáctico, un CSV, un contrato JSON y un script bastan. Para una organización, quizá necesitas catálogo, feature store, lineage estándar, lakehouse y gates en CI. Lo importante es no saltarse la pregunta: **qué evidencia tengo de que este dato sigue siendo el dato que creo que es**.

Una forma sencilla de leer esta arquitectura es seguir una fila. Primero existe en una fuente: un ticket, un documento, una traza. Después entra en una zona controlada mediante ingestión. Luego se valida: tipos, nulos, valores permitidos, licencias. Más tarde se transforma: quizá se limpia, se trocea, se convierte en embedding o se convierte en feature. Finalmente se consume: entrena, evalúa, alimenta un RAG o aparece en un dashboard.

Si cada paso deja evidencia, puedes investigar un fallo con calma. Si no la deja, dependes de recordar quién exportó qué archivo, con qué filtro y en qué día. Eso no escala, y además impide que un alumno aprenda ingeniería reproducible. La meta no es montar una plataforma enorme desde el primer día; la meta es que incluso el proyecto pequeño tenga una línea clara entre fuente, transformación, versión y decisión.



El dato no se queda quieto: se ingiere, valida, transforma, versiona, consume y monitoriza. Cada paso debe dejar evidencia.

De raw a dataset publicable

Para una persona de ingeniería de datos, la palabra “dataset” suele ser demasiado corta. Falta saber en qué zona vive, qué garantías tiene y para qué consumidores está preparado. No es lo mismo un export crudo de una herramienta de soporte que una tabla curada para entrenamiento, un índice de RAG, una matriz de features o una muestra congelada para evaluación.

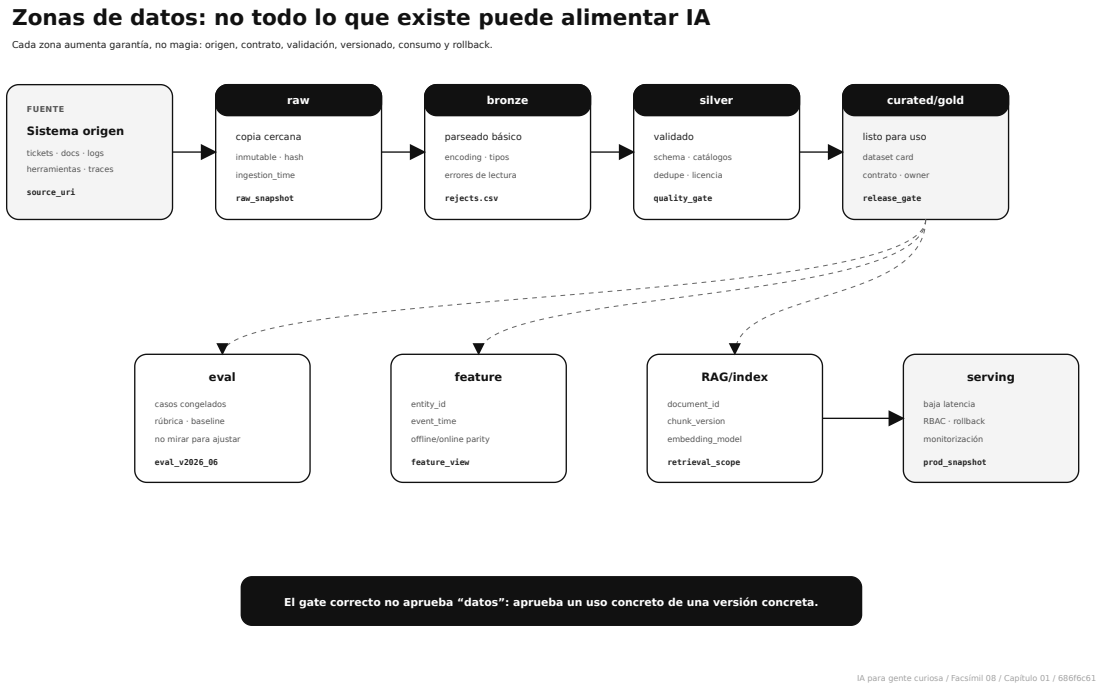
Una arquitectura práctica separa zonas. Los nombres cambian según la empresa (`raw`, `bronze`, `silver`, `gold`, `curated`, `trusted`, `serving`), pero la intención es estable: no mezclar dato recién llegado con dato apto para decidir. Si todo acaba en una misma carpeta, cualquier script puede leer una versión a medias, entrenar con datos sin licencia o evaluar contra una tabla que todavía no pasó checks.

ZONA	QUÉ CONTIENE	QUÉ GARANTÍA DEBERÍA TENER	USO TÍPICO EN IA
`raw`	Copia cercana a la fuente.	Inmutabilidad, timestamp de ingestión, fuente y owner.	Auditoría, reproceso, investigación de incidencias.
`bronze`	Dato parseado mínimamente.	Tipos básicos, partición física, registro de errores de lectura.	Punto de partida para validación.
`silver`	Dato normalizado y validado.	Schema estable, catálogos, deduplicación, contratos mínimos.	Datasets tabulares, features iniciales, corpus limpio.
`gold` o `curated`	Dato preparado para un uso concreto.	Semántica documentada, métricas de calidad, sensibilidad y permisos.	Entrenamiento, RAG, evaluación, reporting.
`feature`	Variables derivadas con entidad y tiempo.	Consistencia offline/online, frescura, versión de definición.	Modelos predictivos y sistemas híbridos.
`eval`	Casos congelados para medir.	Separación de desarrollo, rúbrica, baseline y versión.	Evals, regresiones y comparativas.
`serving`	Material que entra en producción.	Baja latencia, control de acceso, monitorización y rollback.	APIs, RAG online, agentes y decisiones automatizadas.

La distinción importante es la garantía, no el nombre. `raw` no es peor que `gold`; cumple otra función. `raw` debe permitir reconstruir. `gold` debe permitir consumir. `eval` debe proteger la medición. `serving` debe proteger latencia, permisos y rollback. Cuando esas responsabilidades se mezclan, el equipo pierde una de las pocas defensas reales contra el caos.

Piensa en una incidencia real: una respuesta automática cita una normativa antigua. Si solo tienes una carpeta final llamada `dataset_limpio`, la investigación empieza a oscuras. Si tienes zonas separadas, puedes preguntar si la fuente original estaba desactualizada, si la ingesta no recogió la versión nueva, si el chunking generó fragmentos ambiguos, si el índice RAG no se regeneró o si serving estaba apuntando a un snapshot anterior. La arquitectura no es burocracia; es una forma de acortar la investigación cuando algo falla.

Un ejemplo cercano: un equipo exporta tickets académicos cada noche. En `raw` guarda el CSV original con hash y fecha. En `bronze` parsea fechas, normaliza encoding y aparta filas ilegibles. En `silver` valida columnas, catálogos, licencias y sensibilidad. En `gold` crea `support_cases_eval_v2026_06`, con splits, dataset card y contrato. En `serving` no se publica ese CSV entero: se publica solo lo que el sistema necesita, con permisos y monitorización.



Separar zonas evita que un experimento, una eval o un índice RAG consuman datos que todavía no tienen contrato suficiente.

Ingesta incremental, datos tardíos y backfills

La mayoría de datasets reales no llegan completos de una vez. Llegan por ventanas, eventos, cargas parciales, reintentos, correcciones y backfills. Ahí aparece una diferencia que suele pasar desapercibida al principio: **tiempo de evento** no es lo mismo que **tiempo de ingestión**. El tiempo de evento dice cuándo ocurrió el hecho. El tiempo de ingestión dice cuándo llegó a tu pipeline.

El modelo de Dataflow de Akidau y otros puso mucho orden conceptual en esta diferencia al hablar de procesamiento de datos desordenados, ventanas, watermarks, latencia y coste.[20] Para este capítulo no necesitas dominar streaming, pero sí entender la idea: si un ticket de mayo llega en junio, una tabla particionada por `ingestion_time` puede parecer actualizada mientras una métrica por `event_time` cambia el pasado.

CONCEPTO	QUÉ PREGUNTA RESPONDE	ERROR FRECUENTE
`event_time`	¿Cuándo ocurrió realmente el caso?	Usar solo fecha de carga y perder cronología.
`ingestion_time`	¿Cuándo entró al sistema de datos?	Confundir llegada tardía con evento reciente.
Watermark	¿Hasta qué punto creo que una ventana está casi completa?	Cerrar una ventana demasiado pronto.
Datos tardíos	¿Qué hago con eventos que llegan después del cierre?	Reescribir métricas sin registrar versión.
Backfill	¿Cómo recalculo histórico con reglas nuevas?	Ejecutar a mano sin contrato ni manifiesto.
Idempotencia	¿Puedo reejecutar sin duplicar ni corromper?	Insertar dos veces la misma corrección.
CDC	¿Cómo capturo cambios de una fuente transaccional?	Tratar un update como una fila nueva sin estado.

Un ejemplo de clase que sí pasa en empresas: el equipo valida `support_cases_eval_v2026_06` el día 6. El día 8 llega una corrección de 40 tickets creados el día 3. Si la evaluación se diseñó por fecha de evento, esos tickets pertenecen al pasado. ¿Reabres el snapshot? ¿Creas una versión nueva? ¿Los dejas para la siguiente ventana? La respuesta depende del contrato. Lo que no deberías hacer es cambiar silenciosamente el CSV y seguir usando el mismo nombre.

Para backfills, la pregunta central es reproducibilidad. Un backfill serio declara versión de código, versión de contrato, rango temporal, fuente, motivo, owner, outputs esperados y política de comparación. Si solo dice “recalcular histórico”, es muy difícil saber si una métrica cambió porque el mundo cambió o porque cambió la receta.

SITUACIÓN	QUÉ HARÍA UN PIPELINE MADURO
Llega dato tardío menor	Lo marca como late, registra ventana afectada y decide si entra en revisión.
Cambia una regla de limpieza	Crea versión nueva de transformación y backfill con manifiesto.
Se corrige una licencia	Reevalúa usos permitidos y bloquea derivados incompatibles.
Se detecta bug de deduplicación	Reprocesa desde raw y compara conteos antes/después.
Cambia un catálogo de etiquetas	Versiona contrato, política de migración y evals afectadas.

Aquí es donde un ingeniero de datos aporta muchísimo a IA: evita que el modelo sea entrenado, evaluado o monitorizado con una línea temporal falsa.

La respiración importante de esta sección es aceptar que el dato no llega como una fotografía perfecta. Llega como una película con escenas fuera de orden. Los sistemas robustos no niegan ese desorden; lo modelan. Guardan `event_time`, `ingestion_time`, hashes, manifiestos y decisiones de backfill para que una corrección histórica no parezca una mejora espontánea del modelo. Cuando el alumno entienda esto, empezará a desconfiar sanamente de cualquier métrica que no pueda decir con qué ventana fue calculada.

Schema evolution: cambiar sin romper al consumidor

Los datos vivos cambian. Un producto nuevo añade valores al catálogo. Un proveedor renombra una columna. Un equipo cambia el significado de una etiqueta. Una tabla que antes recibía `email` empieza a recibir también `chat`. Esto no es un problema en sí mismo; el problema es que cambie sin contrato.

Schema evolution no significa “aceptar cualquier cosa”. Significa clasificar cambios y decidir cómo migrarlos. Apache Iceberg y Delta Lake documentan mecanismos de evolución de schema y snapshots en tablas lakehouse, pero el principio vale incluso con CSV: si un cambio altera lo que un consumidor entiende, no puede entrar como si fuera un detalle.[21][22]

CAMBIO	TIPO	RIESGO	TRATAMIENTO PROFESIONAL
Añadir columna opcional	Compatible	Consumidores antiguos la ignoran.	Documentar, versionar contrato y añadir test.
Añadir columna obligatoria	Rompiente	Pipelines antiguos fallan o imputan mal.	Nueva versión de contrato y periodo de transición.
Renombrar columna	Rompiente	El consumidor cree que falta un dato.	Doble escritura temporal o migración explícita.
Cambiar tipo	Rompiente	Casts silenciosos, pérdida de precisión, errores.	Migración con validación y backfill.
Ampliar catálogo	Potencialmente compatible	Métricas y evals no comparan igual.	Versionar catálogo y revisar splits.
Cambiar significado	Rompiente semántico	Lo peor: parece igual, pero decide otra cosa.	Nueva versión mayor, nota de migración y revalidación.

Para IA, los cambios semánticos son especialmente peligrosos. Una columna `label` puede seguir llamándose igual y haber cambiado de política. Un campo `resolved` puede pasar de “cerrado por agente” a “cerrado por cualquier canal”. Una fuente RAG puede mantener URL y cambiar normativa. Un embedding puede mantener dimensión y venir de otro encoder. Si no versionas semántica, no tienes dataset: tienes una ilusión de continuidad.

La práctica que recomiendo es simple: cada contrato debe tener versión, compatibilidad esperada, owner, fecha de vigencia y ejemplos. No solo ejemplos felices. También casos frontera. Si un campo cambia de significado, el contrato debe decir qué evals, features, índices y dashboards quedan afectados.

Formatos, tablas lakehouse y coste físico

El formato no es una preferencia estética. Afecta coste, latencia, compatibilidad, schema, compresión, particionado y capacidad de reproducir una versión. CSV y JSONL son excelentes para aprender y revisar a ojo. Parquet y Arrow aparecen cuando el volumen, el rendimiento y la interoperabilidad importan. Las tablas Delta, Iceberg o Hudi añaden metadatos transaccionales, snapshots, evolución y operaciones más propias de lakehouse.[232425]

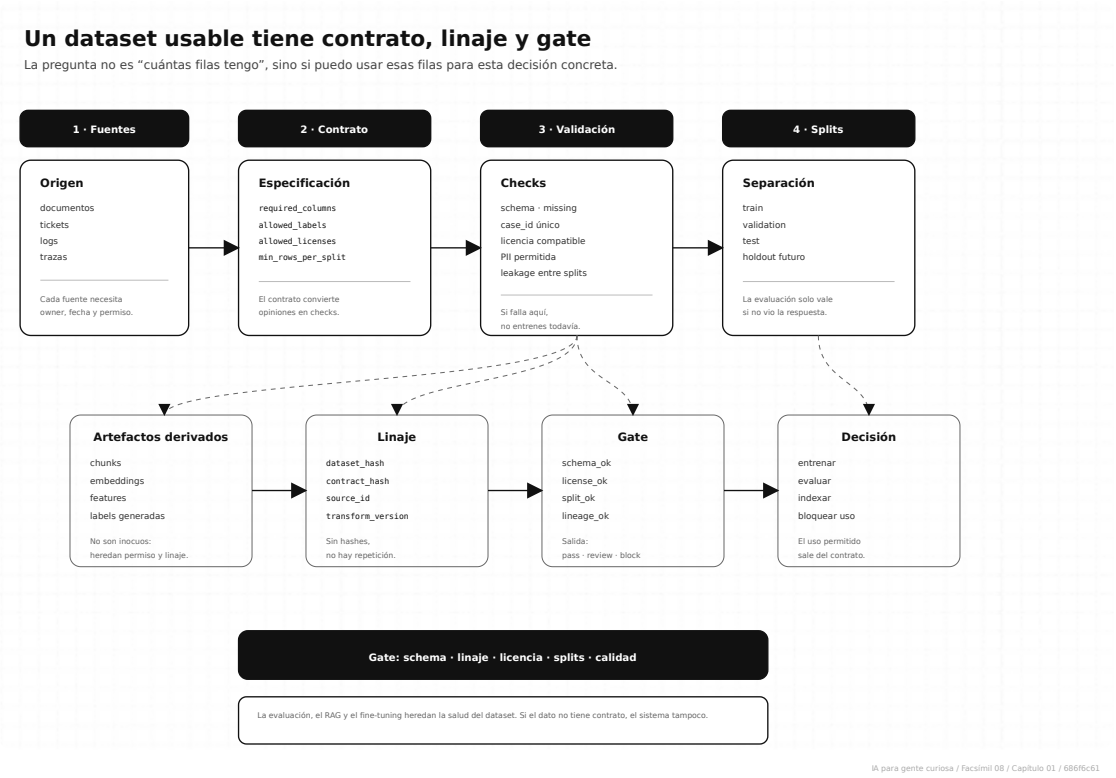
FORMATO O TABLA	APORTA	CUIDADO
CSV	Inspección humana y compatibilidad universal.	Sin tipos fuertes, escapes frágiles, lento para analítica grande.
JSONL	Ejemplos independientes y flexible para LLMs.	Schema menos evidente, campos anidados difíciles de consultar sin motor adecuado.
Parquet	Formato columnar, compresión, lectura por columnas.	Los archivos pequeños degradan rendimiento; el schema debe gobernarse.
Arrow	Formato columnar en memoria e intercambio eficiente.	No sustituye versionado ni contrato de uso.
Delta Lake	Transacciones, schema enforcement, time travel.	Requiere disciplina de tabla, metadatos y operaciones.
Apache Iceberg	Snapshots, evolución de schema/partición, motores múltiples.	Hay que entender catálogo, manifests y estrategia de particionado.
Apache Hudi	Upserts, deletes e incremental processing en data lakes.	Muy útil con cambios frecuentes, pero exige diseño de claves y timeline.

El particionado físico merece una mención aparte. Particionar por fecha suele ser natural, pero no siempre basta. Si consultas siempre por `product` y `event_date`, particionar solo por fecha puede escanear demasiado. Si particionas por demasiadas columnas, puedes crear miles de ficheros pequeños. En IA esto aparece al generar chunks, embeddings o evals: una mala estrategia de archivos puede hacer que un experimento parezca lento por el modelo cuando en realidad el cuello está en I/O.

DECISIÓN FÍSICA	BUENA SEÑAL	MALA SEÑAL
Particionar por `event_date`	Consultas temporales leen pocas particiones.	Llegadas tardías reescriben muchas ventanas sin control.
Particionar por `tenant` o `product`	Aísla dominios o clientes grandes.	Muchos tenants pequeños crean demasiados archivos.
Compactar archivos pequeños	Reduce overhead de planificación y lectura.	Compactar sin manifest rompe reproducibilidad.
Guardar snapshots	Permite comparar y revertir.	Retención infinita sin política de coste.
Usar columnas anidadas	Representa bien JSON/LLM/trazas.	Dificulta tests si el contrato no especifica estructura interna.

Un ingeniero de datos no solo pregunta “¿qué dataset uso?”. Pregunta “¿cómo está almacenado, con qué schema, con qué partición, con qué snapshot, con qué coste de lectura y con qué garantías de evolución?”. Esa pregunta no es de infraestructura: condiciona si la IA se puede sostener.

Anatomía de un contrato de datos para IA



> El contrato de datos convierte una colección de filas en un artefacto que puede usarse, bloquearse, versionarse y auditarse.

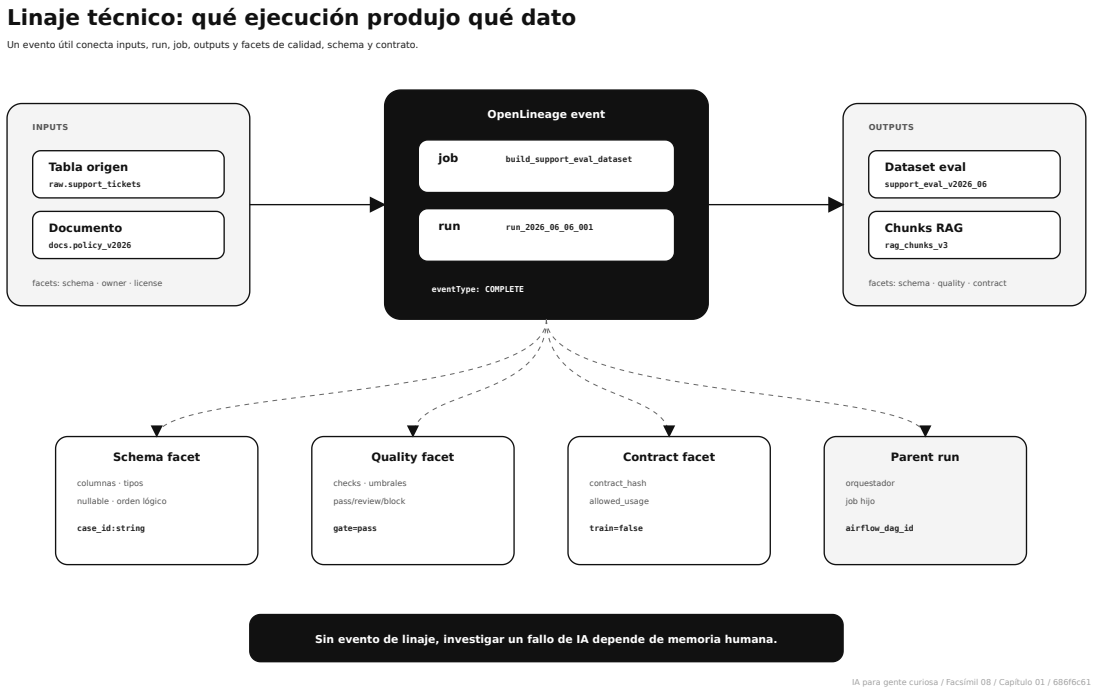
Linaje técnico: del discurso al evento

Decir “tenemos linaje” no basta. Un ingeniero de datos necesita saber qué se captura, cuándo se emite y qué entidades conecta. OpenLineage lo plantea como eventos sobre ejecuciones de jobs: un job tiene una run, esa run tiene inputs, outputs y facets que añaden metadatos.[2627] Esto parece formal, pero resuelve una pregunta muy concreta: si una respuesta de IA falla, ¿qué dataset, contrato, tabla, documento, transformación y versión participaron?

Un evento de linaje útil para IA debería poder decir:

PIEZA	QUÉ REPRESENTA	EJEMPLO APLICADO
`job`	La definición lógica de una tarea.	`build_support_eval_dataset`.
`run`	Una ejecución concreta de ese job.	`run_2026_06_06_001`.
`inputs`	Tablas, ficheros o datasets leídos.	`raw.support_tickets`, `docs.academic_policy`.
`outputs`	Artefactos producidos.	`support_cases_eval_v2026_06`, `rag_chunks_v3`.
`schema facet`	Columnas y tipos del dataset.	`case_id`, `label`, `source_id`, `license`.
`dataQuality facet`	Checks y resultados de calidad.	`missing=0`, `duplicates=0`, `gate=pass`.
`parent run facet`	Quién lanzó una ejecución hija.	Airflow lanza Spark o dbt.
`custom facets`	Metadatos propios del dominio.	`contract_hash`, `dataset_card`, `pii_policy`.

La ventaja para IA es enorme. Si un modelo entrenado el 10 de junio empieza a fallar en becas, no quieres abrir diez notebooks. Quieres poder recorrer el grafo: modelo -> dataset de entrenamiento -> job de construcción -> fuentes -> contrato -> checks -> owner. Si el fallo viene de una fuente obsoleta, el linaje lo hace visible. Si viene de un cambio de schema, también. Si viene de una licencia que no permitía entrenamiento, el problema ya no es técnico solamente: es de gobernanza.



> El linaje técnico conecta el dato con la ejecución que lo produjo. Para IA, eso permite reconstruir qué versión alimentó entrenamiento, RAG, evaluación o producción.

OpenLineage se integra con herramientas como Airflow, Spark, dbt o sistemas propios, pero la idea no depende de una marca. Airflow organiza workflows como DAGs programados y observables; Dagster enfatiza assets como objetos de datos declarados en código; dbt permite tests y contratos en modelos analíticos.[282930] Lo importante para el alumno es saber leer el patrón: definición del asset, ejecución, entradas, salidas, checks, owner y decisión.

Métricas que sí conviene saber

El contrato de datos decide con reglas: `schema_ok`, `lineage_ok`, `license_ok`, `split_ok`, `quality_ok`. Eso es una política de ingeniería. Por eso en el cuaderno del facsímil vive en `contracts/data_contract.json` y se ejecuta como gate.

Las expresiones de esta sección son métricas estándar o proporciones empíricas. Cuando tienen una fuente clásica, la cito. Cuando son checks operativos del cuaderno, aparecen como procedimiento y no como matemáticas.

La tasa de valores faltantes por columna es una proporción empírica. No tiene un autor único: cuenta cuántas filas no traen un valor donde el contrato esperaba uno.

$$miss(c) = \frac{\sum_{i=1}^{n} 1[x_{i,c} = \varnothing]}{n}$$

SÍMBOLO	SIGNIFICADO
$miss(c)$	Proporción de filas donde falta la columna c.
n	Número de filas del dataset.
$x_{i,c}$	Valor de la columna c en la fila i.
1	Indicador: vale 1 si la condición se cumple, 0 si no.

En el cuaderno del facsímil, `support_cases.csv` tiene 18 filas y `max_missing_rate = 0.0`, así que cualquier hueco en una columna obligatoria debería bloquear o revisar el uso. En un dataset real, el umbral puede no ser cero, pero debe estar escrito antes de mirar el resultado.

Para leakage simple entre train y test usamos un check operativo de huellas de texto. El script normaliza cada texto, crea una huella y pregunta si la misma huella aparece en más de un split.

PASO	QUÉ HACE EL CUADERNO	QUÉ SIGNIFICA
Normalizar	Pasa texto a minúsculas y elimina ruido básico.	Evita que una coma esconda un duplicado.
Agrupar por huella	Reúne casos con el mismo texto normalizado.	Busca duplicados textuales.
Comparar splits	Mira si la misma huella aparece en splits distintos.	Detecta una fuga obvia entre entrenamiento y evaluación.
Decidir	Bloquea si el contrato no permite duplicados cruzados.	Protege la métrica de una evaluación contaminada.

Esto no detecta todos los problemas. Un duplicado semántico puede no tener el mismo texto. Pero ya evita una trampa básica: evaluar con respuestas que el sistema pudo ver durante ajuste.

Para duplicados cercanos, una medida clásica es la similitud de Jaccard, propuesta originalmente para comparar conjuntos.[31] En texto, podemos convertir dos ejemplos en conjuntos de tokens o n-gramas y medir cuánto se solapan:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

SÍMBOLO	SIGNIFICADO
A	Conjunto de tokens, palabras o n-gramas del primer ejemplo.
B	Conjunto de tokens, palabras o n-gramas del segundo ejemplo.
$\text{card}(A \cap B)$	Elementos compartidos.
$\text{card}(A \cup B)$	Elementos distintos que aparecen en alguno de los dos.

Si $J(A, B)$ se acerca a 1, los ejemplos son muy parecidos. Eso no prueba por sí solo leakage, porque dos estudiantes pueden preguntar cosas parecidas. Pero sí crea una cola de revisión: si un texto casi igual aparece en train y test, la evaluación puede estar midiendo familiaridad más que generalización.

Para revisar balance de etiquetas podemos usar proporciones:

$$p(y = k) = \frac{n_k}{n}$$

SÍMBOLO	SIGNIFICADO
$p(y = k)$	Proporción de ejemplos con etiqueta k.
n_k	Número de ejemplos de esa etiqueta.
n	Número total de ejemplos.

En el dataset del cuaderno, `answer` aparece 11 veces de 18, así que $p(y = \text{answer}) = 11/18 \approx 0.61$. Si una clase tiene muy pocos ejemplos, el modelo puede aprender a ignorarla y aun así sacar una métrica global aceptable. Por eso conviene mirar distribución de labels, productos, canales e idiomas.

La entropía de Shannon resume cuán repartida está una distribución.[32]

$$H(Y) = -\sum_{k} p(y = k) \log p(y = k)$$

SÍMBOLO	SIGNIFICADO
$H(Y)$	Entropía de la distribución de etiquetas.
$p(y = k)$	Proporción de la etiqueta k.

Con las etiquetas del cuaderno, usando logaritmo natural, la entropía queda alrededor de 0,93 nats. Una entropía baja no es necesariamente mala. Puede significar que el mundo real está desbalanceado. Lo importante es no confundir “realista” con “suficiente para entrenar o evaluar bien”.

Para drift entre una muestra de referencia P y una muestra actual Q, una medida clásica es la distancia de variación total, habitual en teoría de la información y probabilidad.[33]

$$D_{TV}(P, Q) = \frac{1}{2} \sum_i |P_i - Q_i|$$

SÍMBOLO	SIGNIFICADO
P_i	Proporción de la categoría i en la referencia.
Q_i	Proporción de la categoría i en la muestra actual.
D_{TV}	Distancia entre distribuciones categóricas.

El script del cuaderno también calcula Jensen-Shannon, una divergencia simétrica basada en Shannon que compara dos distribuciones mediante una distribución intermedia M.[34]

$$JS(P, Q) = \frac{1}{2} KL(P||M) + \frac{1}{2} KL(Q||M), \quad M = \frac{P + Q}{2}$$

SÍMBOLO	SIGNIFICADO
$JS(P, Q)$	Divergencia Jensen-Shannon entre referencia y muestra actual.
KL	Divergencia de Kullback-Leibler.
M	Distribución media entre P y Q.

En el ejemplo del cuaderno, la columna `product` tiene $D_{TV} = 0.194444$ y $JS = 0.053396$. Eso no bloquea por sí solo, pero ayuda a explicar que producción se está moviendo.

También se usa PSI (*Population Stability Index*) en entornos de scoring y monitorización de carteras; es una métrica industrial habitual en scorecards de riesgo, no una prueba estadística universal.[35]

$$PSI = \sum_i (Q_i - P_i) \log \frac{Q_i}{P_i}$$

SÍMBOLO	SIGNIFICADO
PSI	Índice de estabilidad poblacional.
P_i	Proporción esperada o de referencia.
Q_i	Proporción observada o actual.

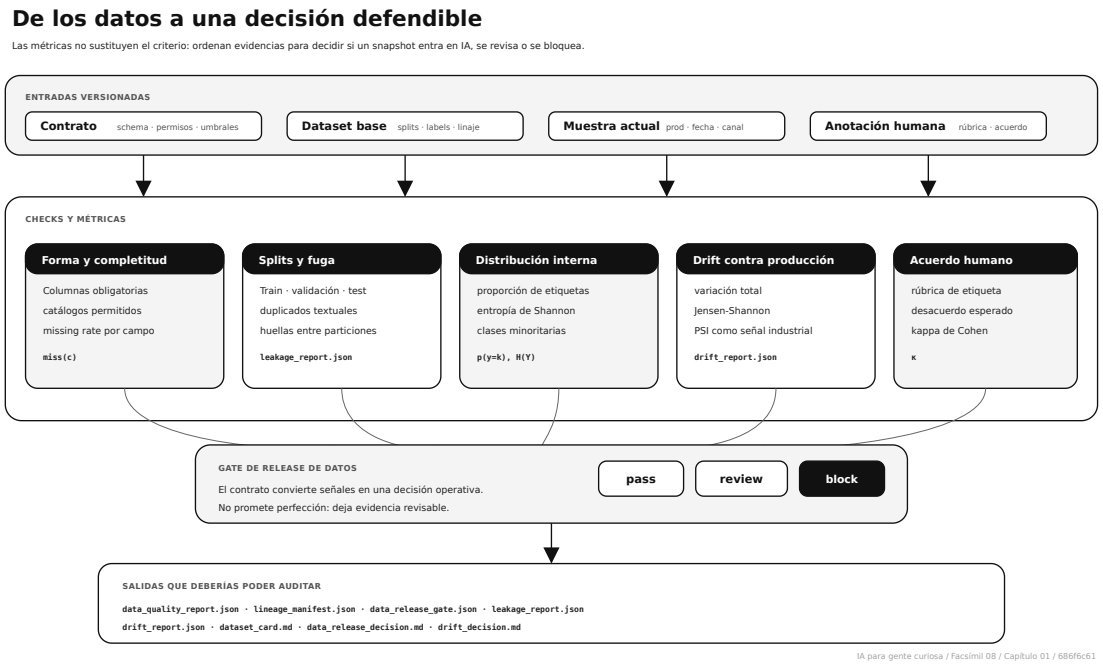
En `product`, el PSI del cuaderno queda en `1.411541` porque una categoría presente en la referencia desaparece en la muestra actual. No lo uses como oráculo. Úsalo como señal: si sube, mira la distribución, el tamaño muestral, la fecha, el origen y si la evaluación sigue representando producción.

Para etiquetas humanas, si dos personas anotan los mismos casos, conviene medir acuerdo. Una versión clásica es kappa de Cohen.[36]

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

SÍMBOLO	SIGNIFICADO
p_o	Acuerdo observado.
p_e	Acuerdo esperado por azar según las distribuciones de anotación.
κ	Acuerdo corregido por azar.

Si dos anotadores coinciden en el 80% de casos y el acuerdo esperado por azar es 50%, entonces $\kappa = (0.80 - 0.50)/(1 - 0.50) = 0.60$. Si las personas no se ponen de acuerdo, quizá el problema no es el modelo. Quizá la etiqueta está mal definida.



El flujo correcto no termina en una métrica bonita: termina en evidencias versionadas y una decisión que otra persona pueda revisar.

Cuando falla un gate: cuarentena, revisión y excepción

Un gate de datos no debería limitarse a gritar “falla” y abandonar al equipo. Si un registro no cumple contrato, el sistema necesita una ruta operativa: apartarlo, explicar por qué, asignar owner, decidir si se corrige, si se descarta o si se permite temporalmente con una excepción documentada. Esto es lo que diferencia un check útil de una traba burocrática.

La salida `block` protege al sistema. La salida `review` protege el criterio humano. La salida `pass` protege la velocidad. Las tres son útiles si tienen consecuencias claras.

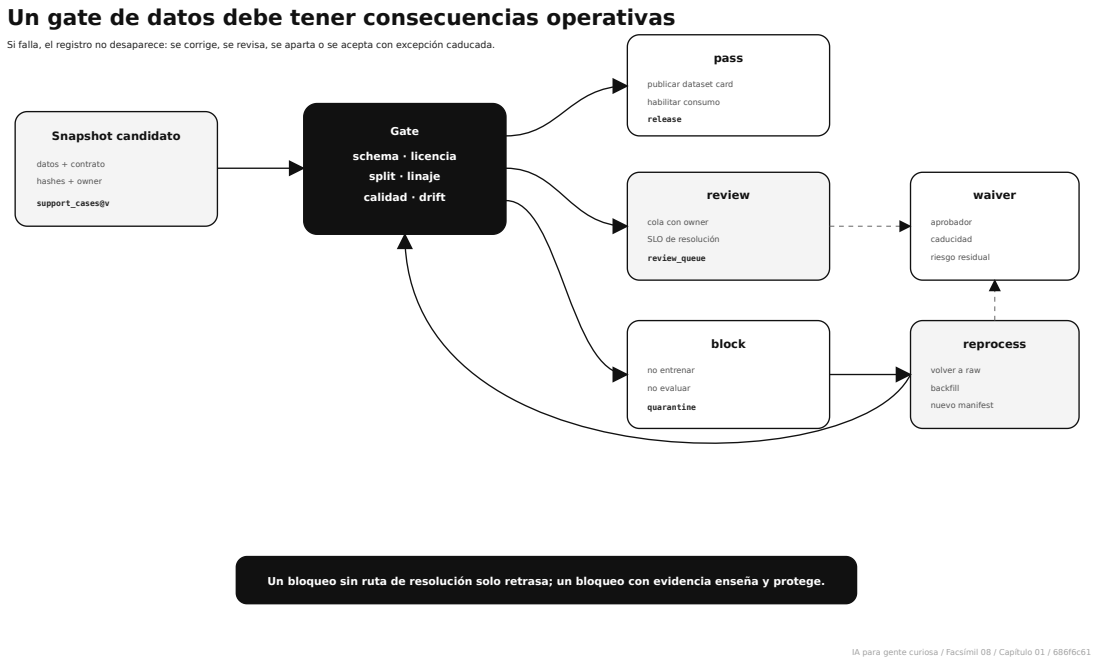
ESTADO	QUÉ SIGNIFICA	QUÉ DEBERÍA OCURRIR
`pass`	El snapshot cumple el contrato para el uso declarado.	Publicar artefactos, manifest y dataset card.
`review`	Hay señal que exige criterio humano o más datos.	Crear cola de revisión con owner y fecha límite.
`block`	Hay una violación incompatible con el uso.	Detener consumo, mandar a cuarentena o reprocess.
`waiver`	Se acepta temporalmente una excepción.	Registrar motivo, aprobador, caducidad y riesgo residual.
`quarantine`	El dato queda apartado del flujo normal.	No entrena, no evalúa, no indexa hasta resolver.

La cola de revisión también se puede medir. Little demostró una relación básica de teoría de colas: $L = \lambda W$, donde L es el número medio de elementos en el sistema, λ la tasa media de llegada y W el tiempo medio de permanencia.[37] En un pipeline de datos esto sirve para una intuición muy concreta: si entran incidencias más rápido de lo que el equipo las revisa, la cola crecerá aunque todos trabajen bien.

$$L = \lambda W$$

SÍMBOLO	SIGNIFICADO EN UNA COLA DE DATOS
L	Registros, chunks, etiquetas o excepciones pendientes de resolver.
λ	Ritmo medio al que entran incidencias al gate.
W	Tiempo medio que una incidencia permanece pendiente.

Ejemplo: si llegan 20 registros a revisión cada día y el tiempo medio de resolución es 3 días, la cola media esperada ronda 60 elementos. No necesitas un sistema complejo para entender el problema: o bajas la entrada de errores, o aumentas capacidad de revisión, o cambias el contrato, o el backlog crecerá.



> El gate tiene que transformar un fallo en una ruta de trabajo: publicar, revisar, bloquear, reprocesar o aceptar una excepción temporal y trazable.

La excepción (`waiver`) merece cuidado. En producción real siempre aparecerá alguien diciendo “déjalo pasar solo esta vez”. A veces tiene sentido: una columna opcional no llega a tiempo, un proveedor corrige mañana, un piloto necesita muestra limitada. Pero una excepción sin caducidad se convierte en política permanente. Por eso debe guardar motivo, aprobador, alcance, fecha de expiración y qué métrica vigilará el riesgo.

Los siete contratos que debería tener un dataset

Un contrato de datos no tiene que ser enorme para ser útil. Debe responder siete preguntas:

CONTRATO	PREGUNTA	EJEMPLO
Schema	¿Qué columnas existen y qué valores son válidos?	`label` solo puede ser `answer`, `ask_more`, `escalate`.
Linaje	¿De dónde viene cada fila?	`source_id`, `created_at`, `owner`, `dataset_hash`.
Uso permitido	¿Para qué puede usarse?	Entrenar, evaluar, consultar, indexar o solo revisar.
Sensibilidad	¿Qué cuidado exige?	Riesgo de PII bajo, medio o alto.
Splits	¿Qué se usa para entrenar y qué para medir?	`train`, `validation`, `test`.
Calidad mínima	¿Qué bloquea uso?	Missing, duplicado, licencia incompatible, etiqueta no permitida.
Decisión	¿Qué hacemos con el resultado?	`pass`, `review`, `block`.

Estos contratos no son burocracia. Son una forma de convertir intuiciones en reglas que se puedan revisar. Si alguien dice “este dataset es bueno”, podemos preguntar: ¿bueno para qué uso?, ¿con qué permisos?, ¿para qué población?, ¿con qué fecha de corte?, ¿con qué separación entre entrenamiento y evaluación? Un contrato no elimina la discusión, pero la vuelve concreta.

También evita que el dato cambie silenciosamente. Si un productor añade una columna, cambia el significado de `label` o deja de enviar `source_id`, el consumidor no debería enterarse semanas después por una caída de calidad. El contrato permite detectar el cambio en el momento en que entra al pipeline.

Esta idea conecta con model cards. Una model card documenta un modelo; una dataset card documenta el material que lo alimenta o lo evalúa.[38] Las dos cosas se necesitan: un modelo sin datos documentados es difícil de interpretar; un dataset sin uso previsto puede acabar aplicado donde no toca.

Privacidad y seguridad: el dato derivado también cuenta

La privacidad no empieza cuando el dato llega al modelo. Empieza en la fuente y continúa en cada derivado. Un chunk, un embedding, una feature, una traza o una etiqueta generada pueden seguir revelando información sensible aunque el texto original ya no esté visible. Por eso el contrato de datos debe hablar de sensibilidad y uso permitido, no solo de columnas.

La regla práctica es minimizar antes de sofisticar. Si el sistema no necesita nombre, DNI, email o teléfono, no deberían entrar en el dataset. Si necesita un identificador estable, quizá basta un identificador pseudónimo. Si necesita agrupar por estudiante, quizá no necesita saber quién es. Y si necesitas conservar el dato sensible para una auditoría, separa acceso, registra consulta y define retención.

CONTROL	QUÉ PROTEGE	EJEMPLO APLICADO
Minimización	No recoger lo que no hace falta.	Eliminar teléfono si el modelo solo clasifica tema.
Pseudonimización	Separar identidad directa del caso.	`student_id_hash` en lugar de DNI.
Tokenización	Sustituir valores sensibles por referencias controladas.	`email_token` resuelto solo en sistema autorizado.
Enmascarado	Reducir exposición en outputs o logs.	Mostrar `***@universidad.es`.
Acceso por columna	Limitar quién puede ver campos concretos.	Equipo de IA ve `label`, no `email`.
Acceso por fila	Limitar por tenant, país, equipo o sensibilidad.	Solo responsables ven `pii_risk=high`.
Retención	Borrar o archivar cuando ya no hay finalidad.	Trazas de prompts se purgan a los 30 días.
Auditoría	Saber quién leyó o exportó datos.	Log de acceso a dataset sensible.

El error típico es pensar que el embedding ya está “anonimizado” porque no se lee como texto. No basta. Un embedding puede permitir búsquedas, inferencias o reconstrucciones parciales según el contexto y los modelos disponibles. Para este capítulo, la conclusión es sencilla: un dato derivado hereda, como mínimo, la sensibilidad y permisos del dato origen hasta que demuestres otra cosa.

También hay que proteger los logs. Muchos sistemas de IA fallan por la puerta más aburrida: un prompt con información sensible queda guardado en una traza, una excepción imprime el payload completo o una cola de revisión exporta casos a una hoja compartida. La ingeniería de datos para IA debe tratar logs, trazas y outputs como datasets.

DERIVADO	PREGUNTA DE PRIVACIDAD
Chunk RAG	¿Puede mostrar texto literal? ¿Tiene licencia y vigencia?
Embedding	¿Hereda permisos del documento? ¿Quién puede buscar sobre él?
Feature	¿Codifica indirectamente un atributo sensible?
Traza	¿Incluye prompt, herramienta, observación o dato personal?
Dataset de eval	¿Contiene casos reales que no deberían circular en clase o demos?
Reporte de error	¿Revela ejemplo sensible en una captura o log?

Una buena práctica: cada dataset sensible debería tener una decisión explícita de uso. No “se puede usar para IA”, sino “se puede usar para evaluación interna hasta esta fecha, sin entrenamiento, con acceso del equipo X, y con outputs agregados”. Ese nivel de precisión evita muchos problemas posteriores.

Herramientas: qué problema resuelve cada una

No hay una herramienta única para “datos de IA”. Hay piezas distintas:

HERRAMIENTA O ESTÁNDAR	RESUELVE	SIRVE CUANDO	NO SUSTITUYE
ODCS	Contrato de datos legible y extensible.	Varios equipos producen y consumen datos.	La validación ejecutable si no conectas checks.
TensorFlow Data Validation	Estadísticas, schema y anomalías.	Quieres perfilar y validar datasets de ML.	Decidir si la licencia o finalidad son correctas.
ML Metadata	Linaje de artefactos, ejecuciones y contextos.	Tienes pipelines ML reproducibles.	Un catálogo de negocio completo.
OpenLineage	Estándar de eventos de linaje de jobs.	Necesitas linaje entre orquestadores y sistemas.	Documentación semántica de cada campo.
DataHub	Catálogo, metadatos, linaje y gobernanza.	Una organización necesita descubrir y gobernar datos.	Checks específicos de entrenamiento si no los defines.
Great Expectations	Expectations y suites de calidad.	Quieres tests de datos repetibles.	Versionado de snapshots o linaje completo.
Feast	Feature store online/offline.	Necesitas features consistentes en train e inferencia.	Limpieza de datos fuente o evaluación de modelo.
Evidently	Drift y monitorización de datos/modelos.	Comparas referencia contra producción.	Arreglar por sí solo una distribución que cambió.
DVC	Versionado de datos y pipelines con Git.	Equipos pequeños o medianos quieren reproducibilidad ML.	Catálogo empresarial o tabla transaccional.
lakeFS	Versionado tipo Git sobre data lake.	Necesitas ramas, commits y rollbacks de datos a escala.	Validación semántica de columnas.
Delta Lake	Tablas con transacciones, schema enforcement y time travel.	Lakehouse con necesidad de historial y rollback.	Contrato de uso por dominio.
Apache Iceberg	Tablas con snapshots y evolución de schema/partición.	Data lake analítico con motores múltiples.	Evaluación de calidad del dataset.
Apache Hudi	Upserts, deletes e incremental processing.	Fuentes que cambian, CDC, correcciones y latencia menor en lakehouse.	Diseño de contrato, claves y calidad.

HERRAMIENTA O ESTÁNDAR	RESUELVE	SIRVE CUANDO	NO SUSTITUYE
Apache Parquet	Formato columnar comprimido.	Analítica grande, lectura parcial de columnas y almacenamiento eficiente.	Versionado, gobernanza o transacciones por sí solo.
Apache Arrow	Formato columnar en memoria e intercambio entre lenguajes.	Pasar datos entre motores, Python, R, JVM o sistemas analíticos.	Persistencia gobernada o contratos semánticos.
Apache Airflow	Orquestación de workflows batch programados.	DAGs con tareas, dependencias, reintentos y calendario.	Entender assets, contratos o calidad si no lo modelas.
Dagster	Orquestación centrada en assets y observabilidad.	Quieres que tablas, modelos o datasets sean objetos declarados.	La política de datos si nadie la escribe.
dbt	Transformaciones SQL, tests y contratos en modelos analíticos.	Warehouse/lakehouse con lógica SQL versionada.	Validar datasets de IA fuera del warehouse si no lo integras.
Hugging Face Datasets	Carga, splits y procesamiento reproducible.	Datasets de NLP, multimodalidad y benchmarks.	Gobernanza interna, licencias y owner.

Una forma práctica de ordenar estas herramientas es por pregunta. Si preguntas “¿qué columnas prometo?”, estás cerca de contratos. Si preguntas “¿cumple lo prometido?”, estás cerca de validación. Si preguntas “¿de dónde viene este resultado?”, estás cerca de linaje. Si preguntas “¿qué versión exacta usé?”, estás cerca de versionado. Y si preguntas “¿ha cambiado producción?”, estás cerca de monitorización.

En un proyecto pequeño no necesitas instalar todo. Puedes empezar con un contrato JSON, scripts de validación, hashes y una dataset card. En un proyecto de empresa, esas mismas ideas pueden acabar repartidas entre DataHub, OpenLineage, Great Expectations, Feast, DVC, lakeFS o tablas lakehouse. La escala cambia; la lógica no.

La lectura correcta para ingeniería no es “elige la herramienta de moda”. Es:

1. ¿Dónde vive la verdad del dataset?
2. ¿Cómo versiono el snapshot?
3. ¿Cómo valido schema y calidad?
4. ¿Cómo registro linaje?
5. ¿Cómo documento finalidad y límites?

6. ¿Cómo detecto que producción cambió?

Si no puedes responder esas seis preguntas, el stack todavía no está maduro aunque tenga nombres modernos. Y al revés: si puedes responderlas con herramientas simples, ya tienes una base didáctica y profesional bastante sana.

Cómo los datos alteran algoritmos

Un dataset no solo alimenta modelos; cambia el comportamiento de los algoritmos.

ALGORITMO O SISTEMA	QUÉ DATO LO CONDICIONA	QUÉ PASA SI EL DATO ESTÁ MAL
Árbol de decisión	Features categóricas, umbrales, missing.	Puede elegir un split cómodo pero espurio.
Regresión logística	Escala, colinealidad, balance de clases.	Pesos inestables o clase minoritaria ignorada.
k-means	Escalado y distancia.	Una variable con rango grande domina los clusters.
Embeddings	Corpus, idioma, longitud y objetivo de entrenamiento.	Vecinos semánticos pobres o sesgo de dominio.
RAG	Chunking, metadatos, vigencia, índice.	Recupera texto irrelevante o caducado.
Fine-tuning	Pares entrada-salida y etiquetas.	Aprende formato superficial o errores de anotación.
Preferencias	`chosen`, `rejected`, rúbrica.	Optimiza gustos mal definidos.
Evaluador LLM	Casos, criterios y ejemplos calibrados.	Premia estilo en vez de verdad operativa.
Agente	Trazas, tools, outcomes y costes.	Aprende trayectorias largas o acciones no útiles.

La parte incómoda es que muchos algoritmos parecen matemáticamente limpios cuando se explican en abstracto, pero se vuelven frágiles al tocar datos reales. Un k-means no “sabe” que una variable está en euros y otra en días; solo ve magnitudes. Una regresión no “sabe” que una clase minoritaria es crítica para negocio; solo optimiza sobre lo que le das. Un retriever no “sabe” que un documento caducó si no lo marcas.

Esto no resta importancia a los algoritmos. Al contrario: para usarlos bien hay que respetar sus supuestos. Si el algoritmo usa distancia, revisa escalado. Si usa etiquetas, revisa criterios de anotación. Si usa ranking, revisa qué documentos pueden aparecer arriba. Si usa trazas, revisa si esas trazas representan la operación real o solo casos cómodos.

Un ejemplo sencillo: si un dataset de soporte tiene casi todo `answer` y muy pocos `ask_more`, un clasificador puede responder siempre y sacar buen accuracy. Pero producto necesita saber cuándo preguntar. En ese caso, el problema no es solo el algoritmo: el dataset no representa suficientemente la decisión que queremos automatizar.

Otro ejemplo: en RAG, dos chunks pueden tener texto parecido, pero uno está vigente y otro caducado. Si no guardas `version` y `expires_at`, el retriever puede hacer lo que le pediste técnicamente y aun así devolver evidencia mala.

La ingeniería de datos para IA consiste justo en esto: mirar dónde se rompen los supuestos antes de culpar al modelo. A veces el modelo falla; muchas otras, el dataset no estaba contando el mundo que creíamos.

Datasets en LLMs: no solo entrenamiento

En sistemas con LLMs, la palabra datos aparece en muchas capas:

CAPA	ARTEFACTO DE DATOS	PREGUNTA DE INGENIERÍA
Pre-entrenamiento	Corpus masivo.	¿Qué mezcla de idioma, dominio y calidad aprende el modelo base?
SFT o ajuste	Pares entrada-salida.	¿Enseñan formato y comportamiento estable?
Preferencias	Comparaciones o rankings.	¿Qué criterio humano o automático se está optimizando?
RAG	Documentos, chunks y metadatos.	¿La fuente está vigente, citada y recuperable?
Embeddings	Vectores derivados de texto o imagen.	¿Se protegen como parte del corpus?
Evals	Casos, referencias y rúbricas.	¿Separan baseline y candidate con evidencia?
Observabilidad	Traces, errores y feedback.	¿Se pueden convertir en regresiones o mejoras del dataset?

Esta tabla ayuda a desmontar una confusión habitual: “datos de LLM” no significa solo corpus de entrenamiento. Un documento de ayuda puede no tocar jamás los pesos del modelo y aun así condicionar una respuesta mediante RAG. Una traza de producción puede no entrenar nada hoy, pero convertirse mañana en un caso de evaluación. Una conversación puede servir para mejorar formato, pero no para actualizar conocimiento vivo.

La clave: **un dato cambia de papel según su uso**. Un documento puede ser fuente RAG, caso de evaluación, ejemplo de fine-tuning o evidencia de una incidencia. Cada uso necesita permiso y contrato.

Por eso el contrato debe hablar de usos, no solo de columnas. La misma fila puede estar permitida para evaluación interna, prohibida para entrenamiento, permitida para recuperación sin mostrar texto literal, o reservada para análisis manual. Sin esa distinción, el dataset se vuelve una bolsa de material disponible y el equipo acaba tomando decisiones por costumbre.

En el día a día

En un equipo de datos, este capítulo aparece en tareas muy concretas. No aparece con el nombre solemne de “gobernanza de datasets”. Aparece cuando una tabla cambia de schema un viernes, cuando un pipeline duplica filas después de un retry, cuando un equipo de producto pregunta por qué bajó una métrica, cuando legal pide saber qué datos entrenaron un modelo o cuando un RAG cita una política antigua.

Un caso típico: soporte académico quiere mejorar un asistente. Producto trae una exportación de tickets. Ingeniería de IA quiere evaluar prompts. Datos pregunta por `case_id`, `source_id`, `event_time`, `license`, `split` y `label_policy_version`. Al principio parece fricción. En realidad está evitando tres errores: entrenar con datos que no se pueden usar, evaluar con duplicados y publicar una métrica imposible de reproducir.

Otro caso: el equipo de datos añade una columna `program_type` porque ahora quiere distinguir grado, máster y doctorado. Si esa columna entra como opcional en `silver`, quizá no rompe nada. Si pasa a ser obligatoria en `gold`, hay que versionar contrato. Si cambia cómo se decide `label`, hay que revisar evaluación. El dato no es solo transporte; es una API entre equipos.

La versión de andar por casa es esta: cuando alguien te mande “el CSV bueno”, no preguntes solo dónde está. Pregunta qué versión es, de qué fuente sale, qué contrato cumple, qué uso permite, qué checks pasaron, qué cambió desde la última vez y qué pasa si producción se mueve.

Por qué debería importarte

Porque un sistema de IA puede fallar aunque el modelo sea bueno. Puede fallar porque el test estaba contaminado, porque el RAG recupera documentos caducados, porque un backfill cambió histórico, porque un embedding se generó con otro encoder, porque un campo cambió de significado o porque una licencia permitía consulta pero no entrenamiento.

Para ingeniería de datos, este capítulo es una forma de entrar en IA sin caer en el teatro del modelo. Tu trabajo no es “preparar datos para que otro entrene”. Tu trabajo es diseñar el suelo sobre el que se podrá entrenar, evaluar, indexar, servir, auditar y corregir. Si ese suelo no tiene contrato, linaje y gates, el resto del sistema queda construido sobre memoria oral.

También importa por coste. Una mala partición física, miles de archivos pequeños, un backfill sin manifiesto o una tabla sin time travel pueden hacer que una evaluación tarde horas, que una comparación no sea repetible o que una incidencia sea imposible de explicar. En IA, el coste técnico se disfraza de “el modelo tarda” o “la eval no cuadra”, pero muchas veces el problema está en datos.

Dónde solía tropezar yo

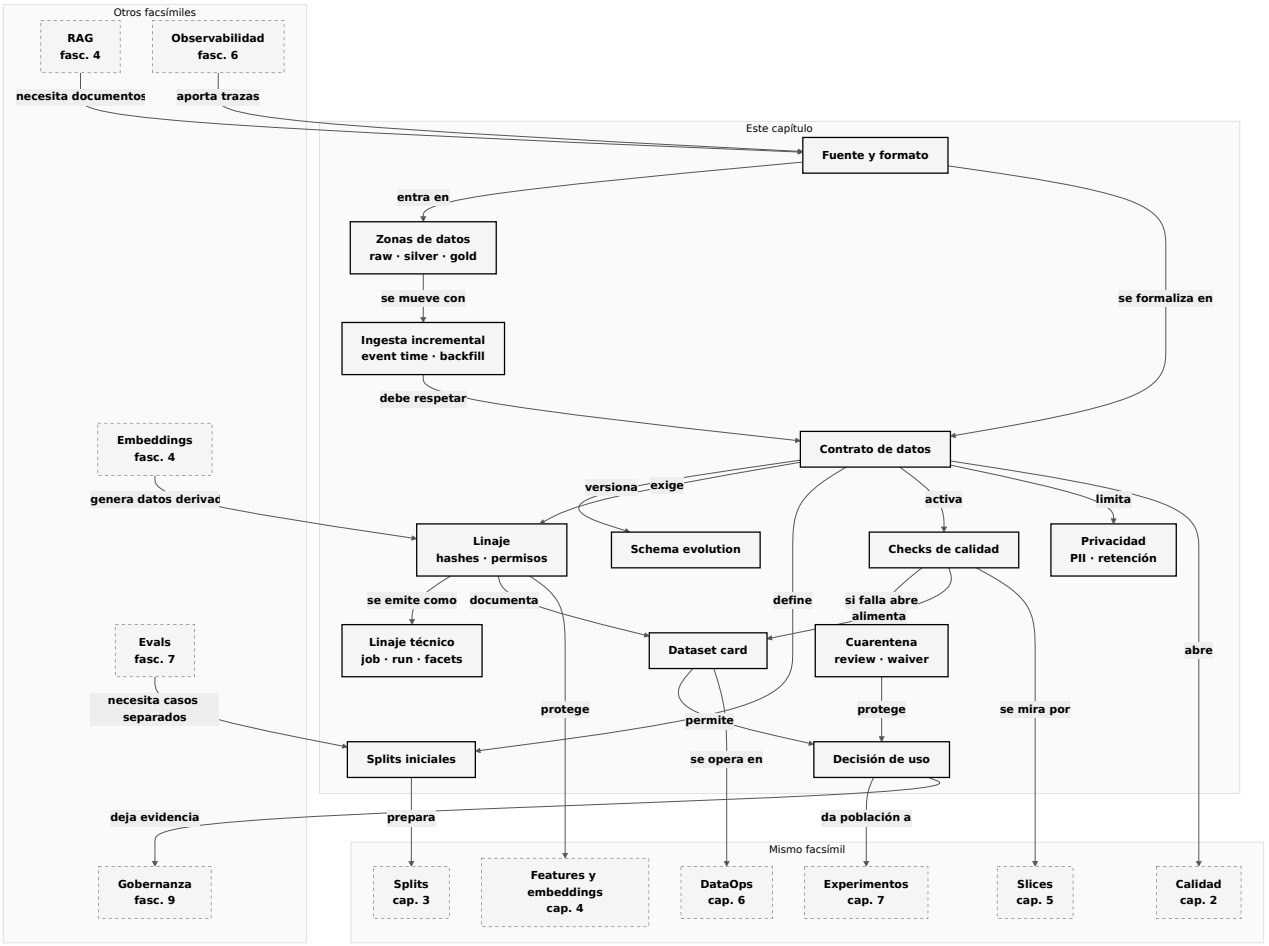
TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Entrenar antes de escribir contrato	El dataset parece obvio porque está delante.	Escribir schema, uso permitido, splits y checks antes de tocar modelo.
Mezclar train y test sin darse cuenta	Duplicados, textos parecidos o casos derivados cruzan particiones.	Calcular huellas y revisar leakage básico.
Usar el mismo dato para RAG, fine-tuning y eval	Todo vive en la misma carpeta y parece “texto útil”.	Declarar finalidad y permiso por fila o documento.
Tratar embeddings como si no fueran datos	Al ser vectores, parecen menos sensibles.	Heredar permisos, linaje y protección del texto origen.
Creer que una licencia vale para todo	Consulta, evaluación y entrenamiento son usos distintos.	Registrar `license` y `consent_scope` por fila.
Guardar preferencias sin política	Un `chosen` sin rúbrica parece verdad universal.	Versionar `preference_policy`, anotador, fecha y criterio.
Documentar solo el modelo	Es más visible presentar una model card.	Añadir dataset card y manifest de linaje.
Limpiar sin versionar	La limpieza mejora algo, pero borra el camino.	Guardar contrato, hashes y decisión de cada snapshot.
Ignorar producción después de aprobar la eval	El test aprobado da tranquilidad falsa durante meses.	Comparar muestras nuevas contra referencia y revisar drift.
Confundir feature store con base de datos normal	Parece solo una tabla más.	Verificar `event_time`, frescura, consistencia offline/online y leakage temporal.
Mezclar raw y curated	Todo parece “el dataset” y cualquier script lee lo que encuentra.	Separar zonas, garantías y usos permitidos.
Recalcular histórico sin manifiesto	El backfill arregla una cosa y cambia tres métricas.	Registrar código, contrato, rango temporal, motivo y outputs.
Aceptar excepciones eternas	El waiver nace temporal y acaba siendo la norma.	Caducidad, owner, riesgo residual y revisión obligatoria.
Tratar Parquet como contrato	El formato guarda columnas, pero no explica semántica.	Añadir contrato, dataset card y tests de datos.

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Guardar trazas sin política de privacidad	Observabilidad se convierte en fuga de datos.	Minimización, redacción, retención y acceso controlado.

Cómo encaja todo

El mapa siguiente conecta este capítulo con lo que ya hemos trabajado. RAG necesita documentos y chunks; embeddings son datos derivados; observabilidad produce trazas; evals necesitan casos separados; interpretabilidad necesita linaje para explicar por qué un sistema decidió algo. Por eso este facsímil no va aparte: es el suelo sobre el que pisan los facsímiles anteriores.

También prepara los siguientes capítulos. Calidad de datos, splits, features, slices, drift y experimentos no son temas aislados. Son capas de una misma pregunta: **qué evidencia tenemos de que el dato representa bien la decisión que queremos tomar**.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Dataset	Conjunto de ejemplos con datos, metadatos y finalidad de uso.
Fuente de datos	Lugar original del que sale el dato: tabla, documento, log, traza o evento.
DatasetDict	Estructura que agrupa splits como `train`, `validation` y `test`.
JSONL	Formato de una línea JSON por ejemplo, muy útil para LLMs, RAG y trazas.
Linaje	Registro de origen, transformación, versión, owner y hash.
Contrato de datos	Especificación verificable de columnas, permisos, splits y checks.
ODCS	Estándar abierto para describir contratos de datos.
Split	Partición destinada a entrenar, validar, probar o evaluar.
Leakage	Fuga de información que contamina la evaluación.
Dataset card	Ficha que documenta finalidad, composición, límites y usos previstos.
Dato derivado	Chunk, embedding, feature, resumen, etiqueta o traza producida desde otro dato.
Feature store	Sistema para mantener features consistentes entre entrenamiento e inferencia.
Drift	Cambio en una distribución respecto a una referencia.
PSI	Índice de estabilidad poblacional usado como señal de drift.
Sensibilidad	Nivel de cuidado por privacidad, licencia, permisos o impacto.
Schema	Forma esperada del dataset: columnas, tipos y valores válidos.
Snapshot	Copia concreta y fechada del dataset.
Time travel	Capacidad de consultar una versión anterior de un dato o tabla.
Hash	Huella criptográfica que identifica una versión exacta.
Owner	Equipo o persona responsable del contrato y sus cambios.
Gate de datos	Decisión automatizada de `pass`, `review` o `block` sobre un dataset.

TÉRMINO	DEFINICIÓN BREVE
Zona raw	Capa donde se conserva el dato cercano a la fuente para poder reconstruir y auditar.
Zona silver	Capa validada y normalizada con schema y calidad mínima.
Zona gold o curated	Capa preparada para un uso concreto: entrenamiento, eval, RAG, reporting o serving.
Watermark	Marca de progreso usada para decidir cuándo una ventana de eventos está suficientemente completa.
Dato tardío	Evento que llega al pipeline después de que su ventana lógica parecía cerrada.
Backfill	Reproceso histórico controlado con versión de código, contrato y manifiesto.
Idempotencia	Propiedad de poder reejecutar sin duplicar ni corromper resultados.
CDC	Captura de cambios de una fuente transaccional: inserts, updates y deletes.
Schema evolution	Gestión de cambios de columnas, tipos, catálogos y significado.
Parquet	Formato columnar comprimido, útil para analítica y lectura parcial de columnas.
Arrow	Formato columnar en memoria para intercambio eficiente entre motores y lenguajes.
Cuarentena de datos	Estado o zona donde se apartan registros que no pueden entrar al flujo normal.
Waiver	Excepción temporal y aprobada que permite seguir con riesgo documentado y caducidad.
Facet de linaje	Metadato asociado a una run, job o dataset en un evento de linaje.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Por qué un dataset no es simplemente una carpeta de filas?
2. ¿Qué diferencia hay entre fuente de datos, dataset y dato derivado?
3. ¿Qué campos mínimos incluirías en un contrato de datos?
4. ¿Por qué una licencia puede permitir RAG pero no fine-tuning?
5. ¿Qué es linaje y qué aporta un hash?
6. ¿Cómo detectarías leakage básico entre train y test?

7. ¿Por qué los embeddings deben tratarse como datos derivados?
8. ¿Qué cambia entre un CSV, un JSONL, un DatasetDict y un corpus RAG?
9. ¿Qué debería contener una dataset card?
10. ¿Qué significa que un gate de datos devuelva `pass`, `review` o `block`?
11. ¿Qué mide PSI y por qué no basta con mirar solo ese número?
12. ¿Qué metadatos necesita un chunk RAG para ser confiable?
13. ¿Qué información mínima debe traer un par de preferencias?
14. ¿Qué problema resuelve un feature store y qué problema no resuelve?
15. ¿Cómo conecta este capítulo con evaluación e interpretabilidad?
16. ¿Qué diferencia hay entre zona raw, silver, gold, eval y serving?
17. ¿Por qué `event_time` e `ingestion_time` no deberían confundirse?
18. ¿Qué registrarías antes de ejecutar un backfill?
19. ¿Qué cambio de schema considerarías compatible y cuál rompiente?
20. ¿Cuándo usarías CSV, JSONL, Parquet, Arrow, Delta/Iceberg/Hudi?
21. ¿Qué partes mínimas esperas en un evento de linaje técnico?
22. ¿Qué harías con un registro que queda en `review` o `block`?
23. ¿Qué significa $L = \lambda W$ en una cola de revisión de datos?
24. ¿Por qué un embedding o una traza también pueden ser datos sensibles?
25. ¿Qué diferencia hay entre un formato físico y un contrato semántico?

En resumen

IDEA	QUÉ TE LLEVAS
Los datos son parte del sistema.	No son una entrada pasiva ni una fase previa menor.
Un dataset necesita contrato.	Schema, linaje, uso permitido, sensibilidad, splits y checks.
La estructura depende de la tarea.	Tabular, JSONL, RAG, preferencias, trazas y features no piden lo mismo.
La evaluación depende del dato.	Si hay leakage o etiquetas débiles, la métrica miente.
RAG y embeddings también son DataOps.	Chunks y vectores heredan permisos, versiones y linaje.
El drift cambia decisiones.	Un test aprobado puede dejar de representar producción.
La documentación importa.	Dataset card, manifest y decisión técnica evitan memoria oral.
La práctica empieza con un gate.	Antes de entrenar, indexar o evaluar, audita el dataset.
Lo profesional es dejar evidencia.	Contrato, hashes, reportes y decisión deben poder repetirse.
La arquitectura separa garantías.	Raw, silver, gold, eval y serving no deben mezclarse.
El tiempo del dato importa.	Event time, ingestion time, watermarks y backfills cambian la lectura.
El formato no es contrato.	Parquet, Arrow o Iceberg ayudan, pero no sustituyen semántica y permisos.
Un gate necesita ruta operativa.	Pass, review, block, cuarentena y waiver deben tener consecuencias claras.
La privacidad se hereda.	Chunks, embeddings, features y trazas pueden seguir siendo sensibles.

Para saber más

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E. y Whittle, S. (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8(12), 1792-1803. https://doi.org/10.14778/2824032.2824076

Apache Arrow. (2026). Apache Arrow Documentation. https://arrow.apache.org/docs/index.html

Apache Hudi. (2026). Apache Hudi Documentation. https://hudi.apache.org/docs/overview/

Apache Parquet. (2026). Apache Parquet Documentation. https://parquet.apache.org/docs/

Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., Koo, C. Y., Lew, L., Mewald, C., Modi, A. N., Polyzotis, N., Ramesh, S., Roy, S., Whang, S. E., Wicke, M., Wilkiewicz, J., Zhang, X. y Zinkevich, M. (2017). TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. *KDD*, 1387-1395. https://doi.org/10.1145/3097983.3098021

Bitol. (2026). Open Data Contract Standard. https://bitol-io.github.io/open-data-contract-standard/latest/

Breck, E., Cai, S., Nielsen, E., Salib, M. y Sculley, D. (2017). The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *IEEE Big Data*, 1123-1132. https://research.google/pubs/pub46555/

DataHub. (2026). DataHub Documentation. https://docs.datahub.com/

dbt Labs. (2026). Model Contracts. https://docs.getdbt.com/docs/mesh/govern/model-contracts

DVC. (2026). What is DVC? https://dvc.org/doc/user-guide/what-is-dvc

Evidently AI. (2026). Data Drift Documentation. https://docs.evidentlyai.com/metrics/explainer_drift

Feast. (2026). Feast Documentation. https://docs.feast.dev/

Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., Wallach, H., Daumé III, H. y Crawford, K. (2021). Datasheets for Datasets. *Communications of the ACM*, 64(12), 86-92. https://doi.org/10.1145/3458723

Great Expectations. (2026). Expectations Overview. https://docs.greatexpectations.io/docs/cloud/expectations/expectations_overview/

Hugging Face. (2026). Datasets Documentation. https://huggingface.co/docs/datasets/

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37, 547-579.

lakeFS. (2026). lakeFS Documentation. https://docs.lakefs.io/

Little, J. D. C. (1961). A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. https://doi.org/10.1287/opre.9.3.383

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D. y Gebru, T. (2019). Model Cards for Model Reporting. *FAT*, 220-229. https://doi.org/10.1145/3287560.3287596

OpenLineage. (2026). OpenLineage Documentation. https://openlineage.io/docs/

Pushkarna, M., Zaldivar, A. y Kjartansson, O. (2022). Data Cards: Purposeful and Transparent Dataset Documentation for Responsible AI. arXiv. https://arxiv.org/abs/2204.01075

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F. y Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. *NeurIPS*. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

TensorFlow. (2026). ML Metadata. https://tensorflow.github.io/tfx/guide/mlmd/

TensorFlow. (2026). TensorFlow Data Validation. https://www.tensorflow.org/tfx/data_validation/get_started/

Notas

1. Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., Wallach, H., Daumé III, H. y Crawford, K. (2021). Datasheets for Datasets. *Communications of the ACM*, 64(12), 86-92. https://doi.org/10.1145/3458723 ↩

2. Pushkarna, M., Zaldivar, A. y Kjartansson, O. (2022). *Data Cards: Purposeful and Transparent Dataset Documentation for Responsible AI*. arXiv. https://arxiv.org/abs/2204.01075 ↩

3. Hugging Face. (2026). *Datasets Documentation*. https://huggingface.co/docs/datasets/. Consultado el 6 de junio de 2026. ↩

4. Sculley, D. y otros (2015). Hidden Technical Debt in Machine Learning Systems. *NeurIPS*. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html ↩

5. Amershi, S. y otros (2019). Software Engineering for Machine Learning: A Case Study. *ICSE-SEIP*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042 ↩

6. Baylor, D. y otros (2017). TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. *KDD*, 1387-1395. https://doi.org/10.1145/3097983.3098021 ↩

7. Breck, E., Cai, S., Nielsen, E., Salib, M. y Sculley, D. (2017). The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *IEEE Big Data*, 1123-1132. https://research.google/pubs/pub46555/ ↩

8. Bitol. (2026). *Open Data Contract Standard*. https://bitol-io.github.io/open-data-contract-standard/latest/. Consultado el 6 de junio de 2026. ↩

9. TensorFlow. (2026). *TensorFlow Data Validation*. https://www.tensorflow.org/tfx/data_validation/get_started/. Consultado el 6 de junio de 2026. ↩

10. TensorFlow. (2026). *ML Metadata*. https://tensorflow.github.io/tfx/guide/mlmd/. Consultado el 6 de junio de 2026. ↩

11. OpenLineage. (2026). *OpenLineage Documentation*. https://openlineage.io/docs/. Consultado el 6 de junio de 2026. ↩

12. DataHub. (2026). *DataHub Documentation*. https://docs.datahub.com/. Consultado el 6 de junio de 2026. ↩

13. Great Expectations. (2026). *Expectations Overview*. https://docs.greatexpectations.io/docs/cloud/expectations/expectations_overview/. Consultado el 6 de junio de 2026. ↩

14. Feast. (2026). *Feast Documentation*. https://docs.feast.dev/. Consultado el 6 de junio de 2026. ↩

15. Evidently AI. (2026). *Data Drift Documentation*. https://docs.evidentlyai.com/metrics/explainer_drift. Consultado el 6 de junio de 2026. ↩

16. DVC. (2026). *What is DVC?*. https://dvc.org/doc/user-guide/what-is-dvc. Consultado el 6 de junio de 2026. ↩

17. lakeFS. (2026). *lakeFS Documentation*. https://docs.lakefs.io/. Consultado el 6 de junio de 2026. ↩

18. Delta Lake. (2026). *Delta Lake Documentation*. https://docs.delta.io/. Consultado el 6 de junio de 2026. ↩

19. Apache Iceberg. (2026). *Apache Iceberg Documentation*. https://iceberg.apache.org/docs/latest/evolution/. Consultado el 6 de junio de 2026. ↩

20. Akidau, T. y otros (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8(12), 1792-1803. https://doi.org/10.14778/2824032.2824076 ↩

21. Apache Iceberg. (2026). *Apache Iceberg Documentation*. https://iceberg.apache.org/docs/latest/evolution/. Consultado el 6 de junio de 2026. ↩

22. Delta Lake. (2026). *Delta Lake Documentation*. https://docs.delta.io/. Consultado el 6 de junio de 2026. ↩

23. Apache Parquet. (2026). *Apache Parquet Documentation*. https://parquet.apache.org/docs/. Consultado el 21 de junio de 2026. ↩

24. Apache Arrow. (2026). *Apache Arrow Documentation*. https://arrow.apache.org/docs/index.html. Consultado el 21 de junio de 2026. ↩

25. Apache Hudi. (2026). *Apache Hudi Documentation*. https://hudi.apache.org/docs/overview/. Consultado el 21 de junio de 2026. ↩

26. OpenLineage. (2026). *Object Model*. https://openlineage.io/docs/spec/object-model/. Consultado el 6 de junio de 2026. ↩

27. OpenLineage. (2026). *Facets & Extensibility*. https://openlineage.io/docs/spec/facets/. Consultado el 21 de junio de 2026. ↩

28. Apache Airflow. (2026). *Apache Airflow Documentation*. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/. Consultado el 7 de junio de 2026. ↩

29. Dagster. (2026). *Dagster Documentation*. https://docs.dagster.io/. Consultado el 21 de junio de 2026. ↩

30. dbt Labs. (2026). *Model Contracts*. https://docs.getdbt.com/docs/mesh/govern/model-contracts. Consultado el 7 de junio de 2026. ↩

31. Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37, 547-579. ↩

32. Shannon, C. E. (1948). A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3), 379-423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x ↩

33. Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley. https://dl.acm.org/doi/book/10.5555/1146355 ↩

34. Lin, J. (1991). Divergence Measures Based on the Shannon Entropy. *IEEE Transactions on Information Theory*, 37(1), 145-151. https://doi.org/10.1109/18.61115 ↩

35. Siddiqi, N. (2006). *Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring*. Wiley. ↩

36. Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1), 37-46. https://doi.org/10.1177/001316446002000104 ↩

37. Little, J. D. C. (1961). A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. https://doi.org/10.1287/opre.9.3.383 ↩

38. Mitchell, M. y otros (2019). Model Cards for Model Reporting. *FAT*, 220-229. https://doi.org/10.1145/3287560.3287596 ↩