Facsímil 08

La ciencia de los datos

Coleccionable completo · 8 capítulos

686f6c61 · 2026

Índice

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

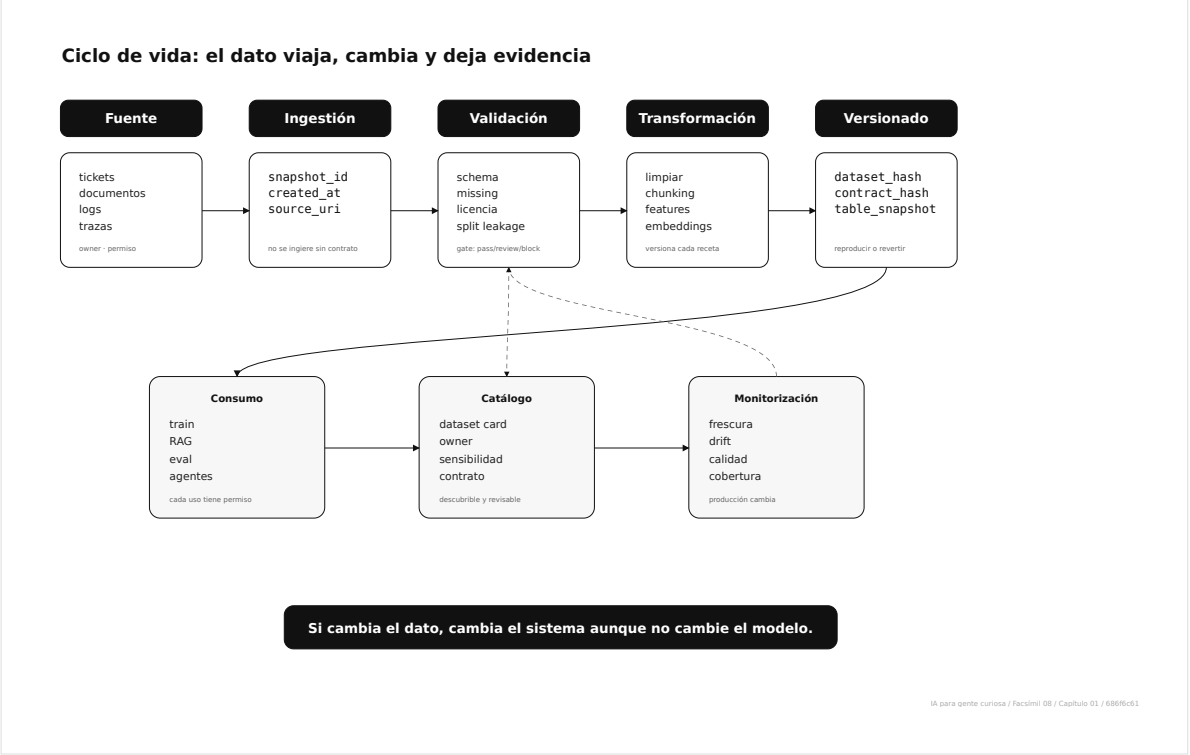
Una arquitectura mínima de datos para IA tiene varias capas. No todas hacen falta en un proyecto pequeño, pero todas deben existir como pregunta mental:

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.

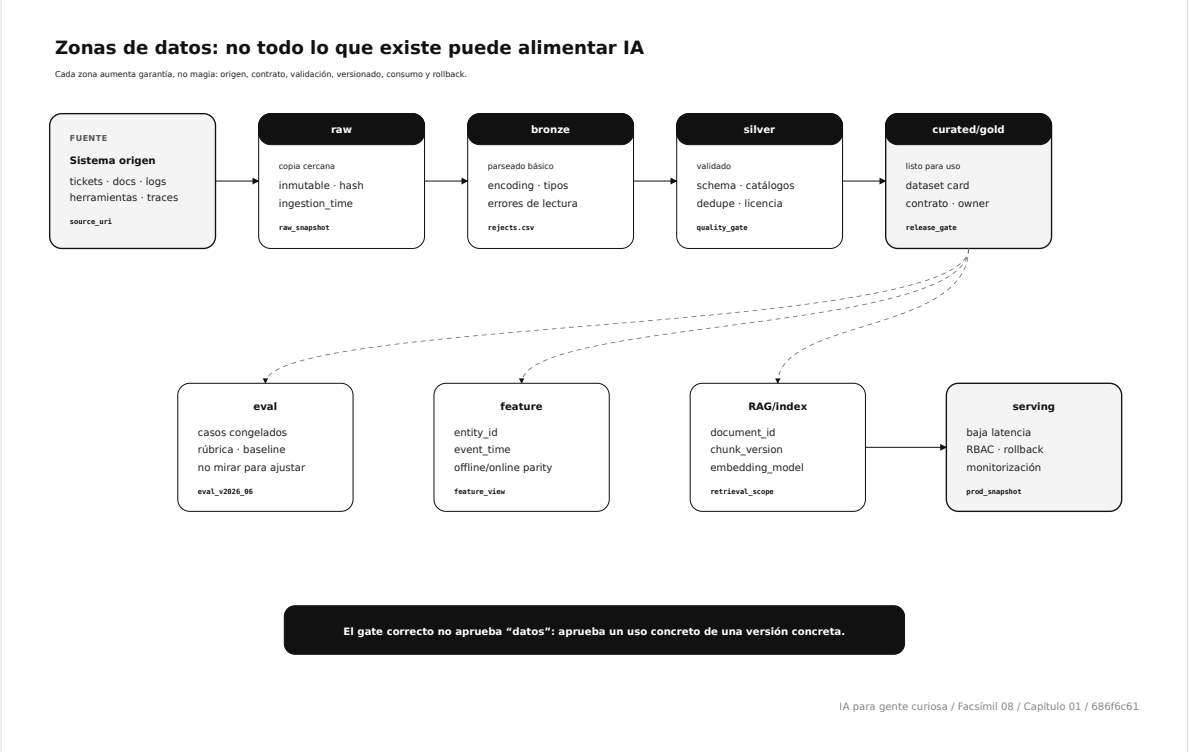
Una arquitectura práctica separa zonas. Los nombres cambian según la empresa (`raw`, `bronze`, `silver`, `gold`, `curated`, `trusted`, `serving`), pero la intención es estable: no mezclar dato recién llegado con dato apto para decidir. Si todo acaba en una misma carpeta, cualquier script puede leer una 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]

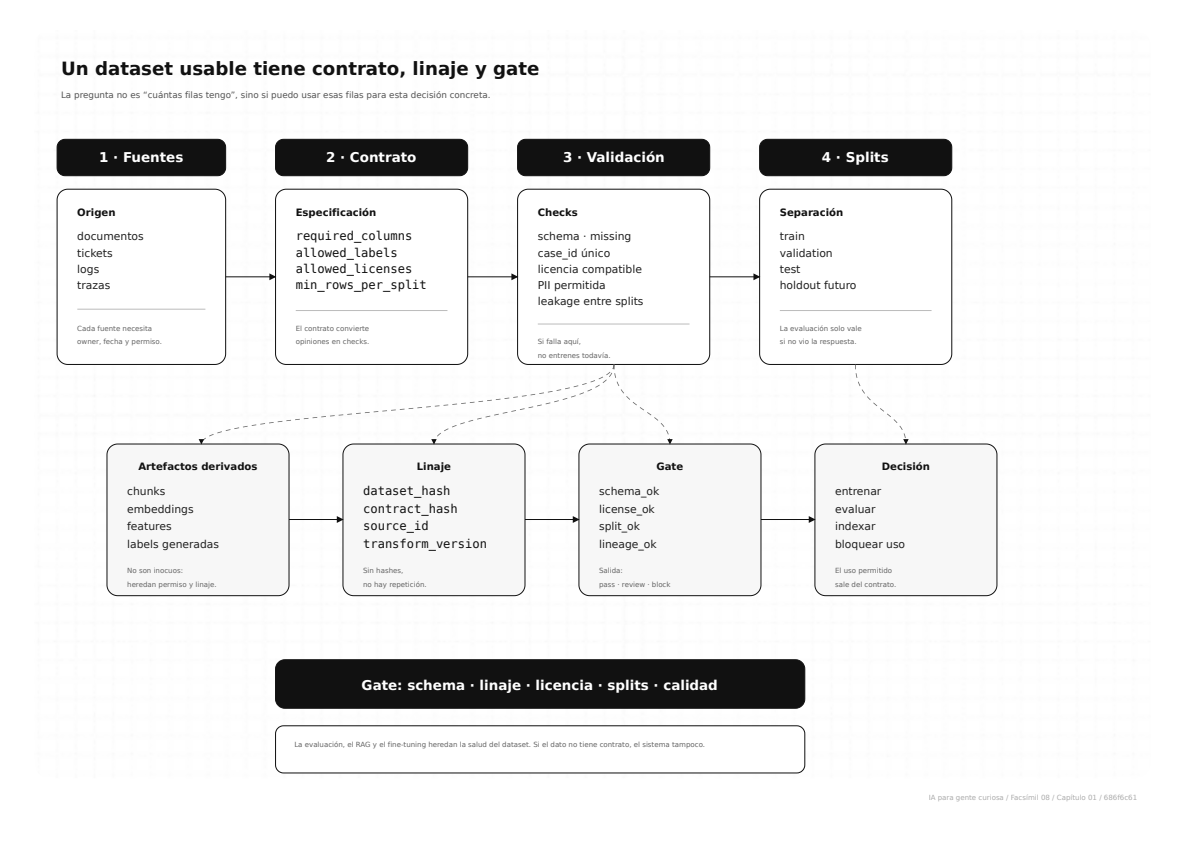
FORMATO O TABLA	APORTA	CUIDADO
CSV	Inspección humana y compatibilidad universal.	Sin tipos fuertes, escapes frágiles, lento para analítica grande.
JSONL	Ejemplos independientes y flexible para LLMs.	Schema menos evidente, campos anidados difíciles de consultar sin motor adecuado.
Parquet	Formato columnar, 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.

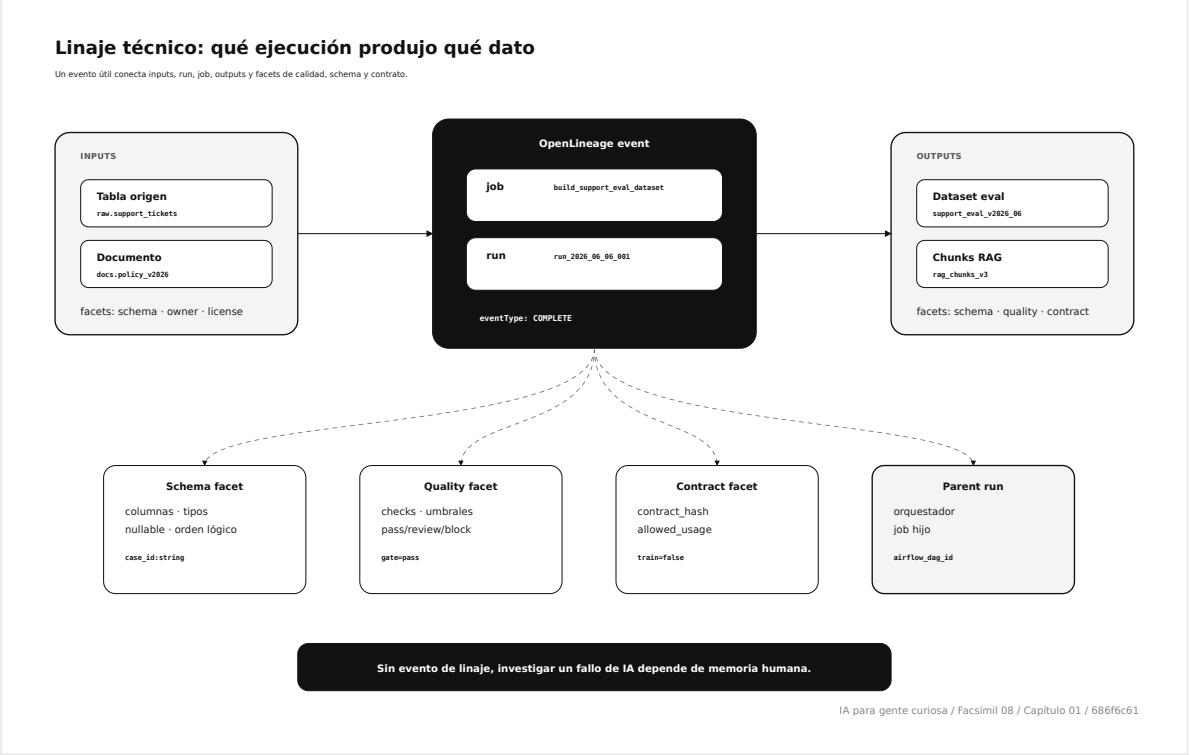
Linaje técnico: del discurso al evento

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 $p(y = \textbf{answer}) = 11/18 \approx 0.61$. 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.

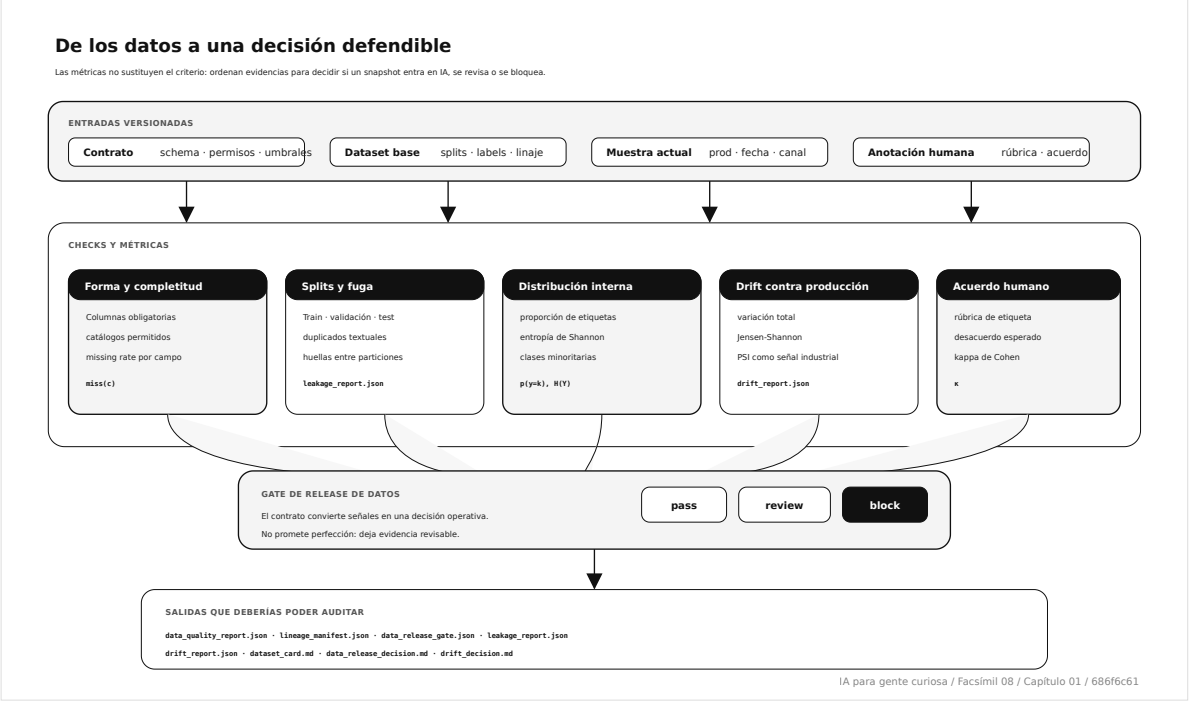
En `product`, el PSI del cuaderno queda en `1.411541` porque una categoría presente en la referencia desaparece en la muestra actual. No lo uses como oráculo. Úsalo como señal: si sube, mira la distribución, el tamaño muestral, la fecha, el origen y si la evaluación sigue representando producción.

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.

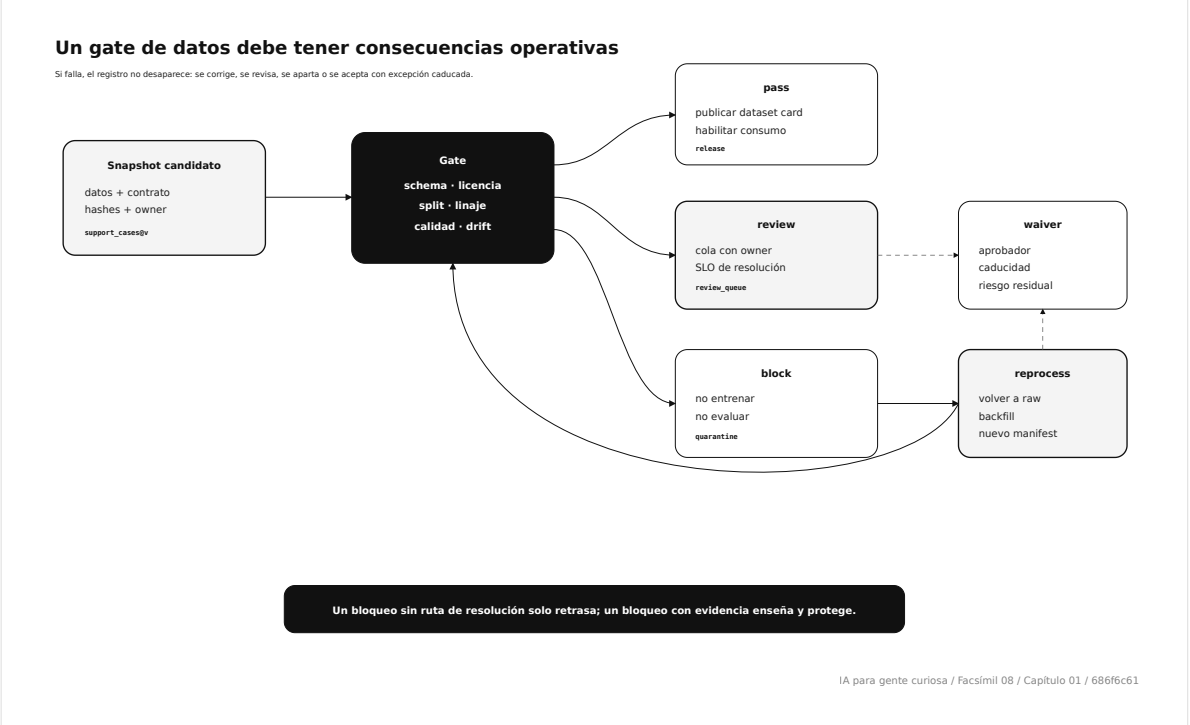
ESTADO	QUÉ SIGNIFICA	QUÉ DEBERÍA OCURRIR
`pass`	El snapshot cumple el contrato para el uso declarado.	Publicar artefactos, manifest y dataset card.
`review`	Hay señal que exige criterio humano o más datos.	Crear cola de revisión con owner y fecha límite.
`block`	Hay una violación incompatible con el 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

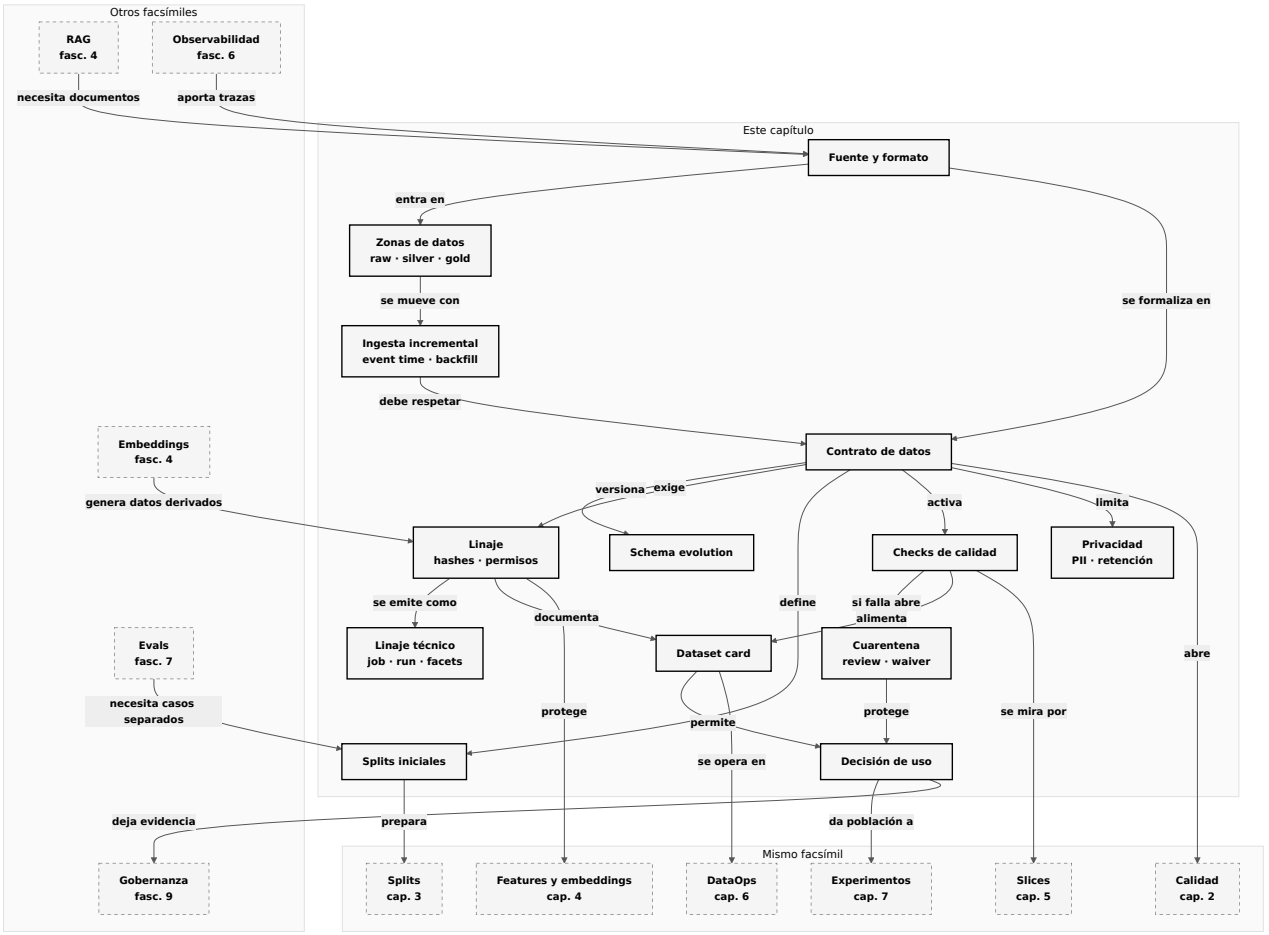
TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Entrenar antes de escribir contrato	El dataset parece obvio porque está delante.	Escribir schema, uso permitido, splits y checks antes de tocar modelo.
Mezclar train y test sin darse cuenta	Duplicados, textos parecidos o casos derivados cruzan particiones.	Calcular huellas 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 ↩

Capítulo 02: Calidad de datos: schema, duplicados, leakage y etiquetas

Entrando en el tema

En datos para IA hay una frase que conviene tatuarse antes de abrir cualquier notebook: **un dataset no falla solo cuando el programa explota**. También falla cuando una etiqueta significa dos cosas distintas, cuando una fila de test se parece demasiado a una de train, cuando una licencia permite evaluar pero no entrenar, cuando una columna nueva entra sin contrato o cuando una cola de revisión crece hasta que nadie sabe qué versión de la verdad está usando el modelo.

La calidad de datos no es una limpieza estética. Es una forma de proteger decisiones. Si el capítulo anterior construía la pregunta “¿de dónde sale este dataset y para qué puedo usarlo?”, este capítulo construye la siguiente: “¿qué pruebas mínimas debe pasar antes de que yo lo use para entrenar, evaluar, recuperar o publicar una conclusión?”.

La respuesta no puede ser una tabla mágica ni una confianza ciega en una herramienta. Tiene que combinar contrato, métricas, revisión humana, evidencia reproducible y criterio de ingeniería. Por eso vamos a hablar de schema, duplicados, leakage y etiquetas como piezas de un mismo sistema: el sistema que evita que una métrica bonita se apoye en datos rotos.

Qué deberías poder hacer al terminar

En el capítulo anterior construimos una idea base: un dataset serio no empieza en el modelo, empieza en el contrato. Ahora damos un paso más. Un contrato escrito no sirve de mucho si no se ejecuta. La calidad de datos aparece justo ahí: en convertir el contrato en comprobaciones que puedan aprobar, revisar o bloquear un dataset antes de que llegue al modelo, al RAG o a una evaluación.

Calidad no significa “datos bonitos”. Significa **datos suficientemente correctos para la decisión que queremos tomar**. Un dataset puede ser aceptable para un prototipo, insuficiente para una evaluación pública, útil para RAG interno y prohibido para entrenamiento. La calidad siempre se lee junto al uso.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Distinguir schema, expectation y contrato.	No confundes “la columna existe” con “el dato es válido”.
Detectar duplicados exactos y cercanos.	Sabes mirar claves repetidas, textos normalizados y similitud.
Explicar leakage con ejemplos de datos.	Puedes detectar cuándo test se contaminó con train o cuándo una feature mira el futuro.
Revisar etiquetas con criterio.	No borras ejemplos difíciles; los llevas a una cola de revisión.
Calcular acuerdo entre anotadores.	Entiendes p_o, p_e y κ.
Diseñar un gate de calidad.	Produces un reporte con `pass`, `review` o `block`.
Conectar calidad con evaluación.	Ves que una métrica puede caer por modelo o por dataset, y sabes separarlo.

La lectura importante es que no basta con detectar fallos: hay que saber qué significan. Un nulo en una columna auxiliar puede abrir revisión; un nulo en `source_id` puede impedir auditar una cita; una etiqueta fuera de catálogo puede romper entrenamiento; un duplicado cruzando train y test puede invalidar una evaluación. El mismo síntoma cambia de gravedad según el uso que vaya a tener el dataset.

La idea central es esta:

> La calidad de datos no se inspecciona al final. Se prueba antes de decidir.

La escena: el modelo parece peor, pero el dataset cambió

Imagina que tenemos una eval de soporte académico. Durante semanas el asistente funciona razonablemente bien. Un día, tras cambiar algunos datos, la métrica baja. El primer impulso es pensar que el modelo ha empeorado o que el prompt ya no sirve.

Pero al mirar el dataset aparecen señales menos espectaculares y más importantes: una etiqueta nueva llamada `resolve` se coló en un catálogo donde solo existían `answer`, `ask_more` y `escalate`; un caso de `train` aparece casi igual en `test`; un campo `source_id` está vacío; una fila de validación lleva licencia de entrenamiento; y dos anotadores no se ponen de acuerdo en varios ejemplos.

Nada de eso requiere cambiar el modelo. Requiere parar la cadena y preguntar: **¿este dataset puede sostener la decisión que estamos tomando?** Si no puede, la métrica deja de ser una medida limpia. Se convierte en una mezcla de comportamiento del modelo, fallos de contrato, errores de etiqueta y contaminación entre splits.

Qué no es calidad de datos

Calidad de datos no es “no tener nulos”. Los nulos importan, pero son solo una parte pequeña. Un dataset puede no tener ningún valor vacío y aun así estar mal: etiquetas inconsistentes, licencia incompatible, duplicados entre splits, texto caducado, clases desbalanceadas o columnas con significado ambiguo.

Tampoco es “pasar un script de limpieza”. Limpiar sin contrato puede empeorar el problema. Si borras todos los casos difíciles porque ensucian la métrica, quizá destruyes justo los ejemplos que el sistema necesita aprender o evaluar. Si deduplicas sin mirar splits, puedes eliminar evidencia legítima o dejar contaminación.

Y no es una puntuación universal. La misma tabla puede estar bien para explorar, mal para entrenar y completamente inaceptable para evaluar. Por eso la pregunta correcta no es “¿tiene calidad?”, sino “¿tiene calidad suficiente para este uso, con este contrato y esta fecha de corte?”.

MALENTENDIDO	LECTURA DE INGENIERÍA
“Si no hay nulos, está limpio”.	Faltan catálogo, licencias, duplicados, leakage, etiquetas y linaje.
“Un duplicado siempre se borra”.	Primero hay que saber si es repetición real, evento legítimo o contaminación.
“La etiqueta registrada es verdad”.	La etiqueta es una decisión humana o automática que puede fallar.
“Si la métrica baja, el modelo empeoró”.	Puede haber cambiado el dataset, el split o la distribución.
“Calidad es cosa de datos, no de ingeniería”.	Un gate de calidad es parte del pipeline de software.

Qué sí es calidad de datos para IA

En este facsímil llamaremos calidad de datos al grado en que un dataset cumple las condiciones necesarias para una decisión de IA: entrenar, evaluar, indexar, recuperar, monitorizar o revisar.

Es una decisión de ingeniería basada en contrato. Para que un dataset avance, el equipo debe poder contestar estas preguntas con evidencias, no con intuiciones:

PIEZA DEL CONTRATO	PREGUNTA QUE DEBE CONTESTAR	EVIDENCIA MÍNIMA
Schema	¿La forma del dataset es la esperada?	Columnas obligatorias, tipos y columnas extra.
Valores	¿Los campos respetan catálogos y rangos?	Etiquetas permitidas, productos válidos, nulos medidos.
Linaje	¿Sabemos de dónde viene cada fila?	`source_id`, fecha, owner, hash y versión.
Licencia	¿El uso previsto está permitido?	Permisos por split: entrenar, evaluar, recuperar o revisar.
Splits	¿La evaluación está separada de aprendizaje y ajuste?	Train, validation y test sin contaminación evidente.
Etiquetas	¿Las referencias tienen criterio estable?	Acuerdo entre anotadores, política de etiqueta y cola de revisión.

La lectura práctica es simple: si una pieza crítica falla, el dataset no debería avanzar como si nada. No hace falta inventar una ecuación para verlo. Hace falta un contrato, checks ejecutables y una decisión trazable.

Ese contrato también protege al equipo de un sesgo muy humano: arreglar lo que se ve fácil y dejar intacto lo que duele. Es cómodo corregir formatos o borrar nulos; es más incómodo revisar etiquetas, licencias, duplicados semánticos o leakage temporal. Pero en IA, muchas métricas falsas nacen precisamente en esas zonas incómodas. Por eso el gate debe guardar evidencia, no solo un estado final.

Dimensiones de calidad: no todo fallo es del mismo tipo

Una forma madura de hablar de calidad de datos es separar dimensiones. Wang y Strong propusieron una idea que sigue siendo útil: la calidad no se define solo desde la tabla, sino desde quien consume el dato y la tarea que necesita resolver.[1] Para IA esto es especialmente importante, porque el mismo campo puede ser correcto para analítica y peligroso para evaluación.

Imagina una columna `created_at`. Si está vacía, falla completitud. Si tiene texto donde esperamos una fecha ISO, falla validez. Si dice que el caso se creó después de su resolución, falla consistencia. Si llega tres días tarde al pipeline, falla actualidad. Y si el campo existe

pero no estaba disponible en el momento de decisión, puede introducir leakage temporal. Es la misma columna, pero no el mismo problema.

DIMENSIÓN	QUÉ PREGUNTA HACE	EJEMPLO EN IA
Completitud	¿Falta algo que el contrato exige?	`source_id` vacío impide auditar una respuesta RAG.
Validez	¿El valor respeta tipo, formato, rango o catálogo?	`label=resolve` no está en el conjunto permitido.
Consistencia	¿Dos piezas de información se contradicen?	`split=test` con licencia solo permitida para entrenamiento.
Unicidad	¿La misma entidad aparece más veces de las permitidas?	`case_id=q004` repetido.
Actualidad	¿El dato está vigente para la fecha de corte?	Documento académico actualizado después de la consulta evaluada.
Representatividad	¿Cubre los casos donde vamos a decidir?	Casi no hay ejemplos de `escalate` en test.
Trazabilidad	¿Podemos reconstruir origen, versión y transformación?	Falta `source_id`, hash o política de anotación.
Adecuación al uso	¿Sirve para entrenar, evaluar, indexar o decidir?	Una muestra exploratoria no debería usarse como benchmark.

Esta tabla evita un error común: decir “calidad baja” y quedarse ahí. En ingeniería necesitamos saber qué dimensión falla, qué uso afecta, si bloquea o abre revisión y qué evidencia queda para repetir la decisión.

Fecha de corte del estado del arte

Fecha de corte: 6 de junio de 2026.
Fuentes consultadas: Wang y Strong sobre dimensiones de calidad, TensorFlow Data Validation, Great Expectations, Pandera, Deequ, Soda, Cleanlab, Snorkel, Evidently, scikit-learn, Jaccard, Levenshtein, Broder, Shannon, Manning y otros y literatura clásica sobre acuerdo entre anotadores.

La idea estable es que la validación de datos debe ser declarativa, reproducible y cercana al pipeline. TensorFlow Data Validation perfila datasets, infiere schema y detecta anomalías contra expectativas.[2] Su referencia de anomalías muestra que un sistema de validación no

solo dice “hay error”, sino que clasifica problemas como tipo inválido, dominio inesperado, valor ausente o cambio de distribución.[3]

Great Expectations organiza la calidad como expectations: afirmaciones verificables sobre columnas, rangos, nulos, formatos o relaciones entre datos.[4] Pandera lleva esa misma filosofía al mundo de DataFrames, con schemas programáticos para validar tipos y propiedades de columnas.[5] Deequ propone “unit tests for data” sobre Spark, una frase muy útil porque acerca la calidad de datos a una práctica que cualquier ingeniero de software reconoce.[6]

Para etiquetas, confident learning propone estimar incertidumbre y errores de etiqueta en datasets, y Cleanlab lo popularizó como herramienta práctica.[7] El trabajo sobre errores de etiqueta en test sets mostró algo incómodo: incluso benchmarks muy usados pueden contener errores de etiqueta suficientes para desestabilizar conclusiones.[8]

Snorkel es importante porque enseña otra vía: crear datos de entrenamiento con supervisión débil mediante funciones de etiquetado, y después combinar señales ruidosas de forma sistemática.[9] La lección para este capítulo no es “usa una herramienta concreta”. Es más básica: si una etiqueta puede fallar, la etiqueta también necesita ingeniería.

Las tripas de la calidad: de schema a decisión

La calidad empieza con schema, pero no termina ahí. El schema responde a la forma: qué columnas existen, qué tipos tienen, qué valores son válidos. Después vienen reglas de contenido: nulos, catálogos, rangos, licencias, duplicados, distribución de clases, coherencia entre anotadores y leakage entre splits.

En el cuaderno del facsímil, cada expectation se trata como un check con tres campos: nombre, severidad y evidencia. No estamos demostrando un teorema; estamos construyendo una puerta de release para datos.

CAMPO DEL CHECK	QUÉ GUARDA	EJEMPLO
Nombre	Qué regla se evalúa.	`label_values`.
Severidad	Qué pasa si falla.	`block` o `review`.
Evidencia	Qué casos explican el fallo.	Filas con `label=resolve`.

El gate del cuaderno agrupa esos checks y les asigna una acción. En otro equipo podrían cambiar los nombres o los umbrales, pero la idea operativa es la misma: no todo fallo pesa igual.

La tabla concreta cuándo devuelve cada estado:

RESULTADO	QUÉ SIGNIFICA	QUÉ HARÍA UN EQUIPO
`block`	Hay fallos que impiden usar el dataset.	No entrenar, no evaluar, no indexar ese snapshot.
`review`	No hay bloqueo técnico, pero hay dudas que requieren criterio.	Abrir cola de revisión y documentar decisión.
`pass`	Cumple el contrato actual para el uso declarado.	Usarlo y guardar reporte/versionado.

La parte importante es la palabra “actual”. Un `pass` no significa que el dataset sea perfecto para siempre. Significa que, bajo este contrato y este uso, no hemos encontrado fallos que impidan avanzar.

Esta distinción evita dos extremos igual de malos. El primero es bloquear todo por miedo, hasta que ningún dataset sirve para aprender. El segundo es aprobar todo porque “ya corregiremos después”. Un gate profesional no pretende purificar los datos; pretende tomar una decisión proporcional: bloquear lo que rompe la validez, revisar lo que necesita criterio humano y aprobar lo que cumple para el uso declarado.

Missing y valores fuera de catálogo

El missing se calcula aquí como proporción muestral de indicadores de ausencia sobre las celdas obligatorias. Es una forma directa de resumir datos faltantes; la literatura clásica de missing data insiste en que, además de la tasa, importa el mecanismo que produjo esos huecos.[10]

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
n	Número de filas.	19 casos.
C	Columnas obligatorias.	`case_id`, `source_id`, `label`, `text`, etc.
$x_{i,c}$	Valor de la columna c en la fila i.	`source_id` de `q007`.
1	Indicador: 1 si falta, 0 si no.	Vale 1 si `source_id` está vacío.
$miss(D)$	Tasa total de missing.	0.003096 en el cuaderno.

Un missing bajo puede seguir siendo grave. Si falta una columna secundaria quizá es revisable; si falta `source_id`, perdemos linaje. Por eso la tasa agregada no basta: hay que mirar qué columna falta y para qué uso era necesaria.

Los valores fuera de catálogo son otra familia de errores. Si `label` solo permite `answer`, `ask_more` y `escalate`, una etiqueta `resolve` no es una variación inocente: rompe la semántica del dataset. Quizá representa una nueva clase legítima, pero entonces el contrato debe cambiar y la evaluación debe rehacerse.

Duplicados exactos y duplicados cercanos

Un duplicado exacto puede detectarse normalizando texto:

PASO	QUÉ HACE	EJEMPLO
Normalizar	Minúsculas, quitar signos y compactar espacios.	`Pago duplicado!!!` pasa a `pago duplicado`.
Comparar	Mira si dos textos normalizados coinciden.	`q001` y `q017`.
Decidir	Si cruzan splits, el fallo puede bloquear evaluación.	Duplicado en train y test.

Pero muchos duplicados no son idénticos. Cambian una palabra o añaden un adjetivo. Para enseñar la idea sin dependencias externas, el cuaderno usa Jaccard sobre conjuntos de tokens. La similitud de Jaccard es una medida clásica basada en intersección y unión de conjuntos.[11]

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
A	Tokens del primer texto.	`consulta`, `documentacion`, `beca`.
B	Tokens del segundo texto.	`consulta`, `documentacion`, `beca`, `necesaria`.
$\text{card}(A \cap B)$	Tokens comunes.	Palabras compartidas.
$\text{card}(A \cup B)$	Tokens totales únicos.	Palabras distintas entre ambos textos.
$J(A, B)$	Similitud de Jaccard.	0.833333 en un par del cuaderno.

Jaccard no entiende semántica profunda. No sabe que “matrícula” y “inscripción” pueden estar relacionadas. Pero es suficiente para enseñar una idea crítica: si un caso muy parecido aparece en train y test, la evaluación deja de ser limpia.

Distancia de edición: cuando cambia una palabra, una letra o un formato

Jaccard mira conjuntos de tokens. Eso es útil para frases parecidas, pero no siempre detecta errores de escritura, identificadores casi iguales o pequeñas variaciones de formato. Para ese caso existe una medida clásica: la distancia de edición de Levenshtein, que calcula cuántas operaciones mínimas hacen falta para convertir una cadena en otra.[1][2]

La definición habitual se escribe como programación dinámica:

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \\ d_{i-1,j-1} + 1[a_i \neq b_j] \end{cases}$$

SÍMBOLO	SIGNIFICADO	LECTURA PRÁCTICA
a_i	Carácter i de la primera cadena.	Una letra del primer texto normalizado.
b_j	Carácter j de la segunda cadena.	Una letra del segundo texto normalizado.
$d_{i,j}$	Coste mínimo hasta esas posiciones.	Cuántas ediciones llevamos acumuladas.
$+1$	Insertar o borrar.	Falta o sobra un carácter.
$1[a_i \neq b_j]$	Sustituir si los caracteres difieren.	Cambiar una letra por otra.

En un dataset de soporte, Levenshtein ayuda a detectar `doc-mat-001` frente a `doc-mat-OO1`, o “matricula ordinaria” frente a “matrícula ordinaria” tras normalizar acentos. No sustituye a Jaccard ni a embeddings. Es otra lente. La regla de ingeniería es combinar lentes según el daño: claves y códigos pueden necesitar distancia de edición; textos largos pueden necesitar n-gramas, Jaccard o embeddings; documentos completos pueden necesitar hashes, similitud semántica y revisión humana.

Duplicados a escala: blocking, MinHash y candidatos

En un CSV de 19 filas puedes comparar todos los pares. En un dataset real, no. El número de pares posibles crece como:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

Con $n = 1\,000\,000$, eso son casi quinientos mil millones de comparaciones. Aunque cada comparación fuera barata, el planteamiento ya nació mal. Por eso los sistemas de calidad de datos no suelen buscar duplicados cercanos comparando todo contra todo. Primero reducen candidatos.

La estrategia clásica tiene tres capas:

CAPA	QUÉ HACE	EJEMPLO
Blocking	Agrupa por una clave barata.	Mismo producto, misma fuente, mismo prefijo de documento.
Fingerprinting	Resume el texto o documento en una firma compacta.	Hashes de shingles o tokens normalizados.
Scoring	Calcula similitud solo entre candidatos.	Jaccard, Levenshtein, embeddings o revisión humana.

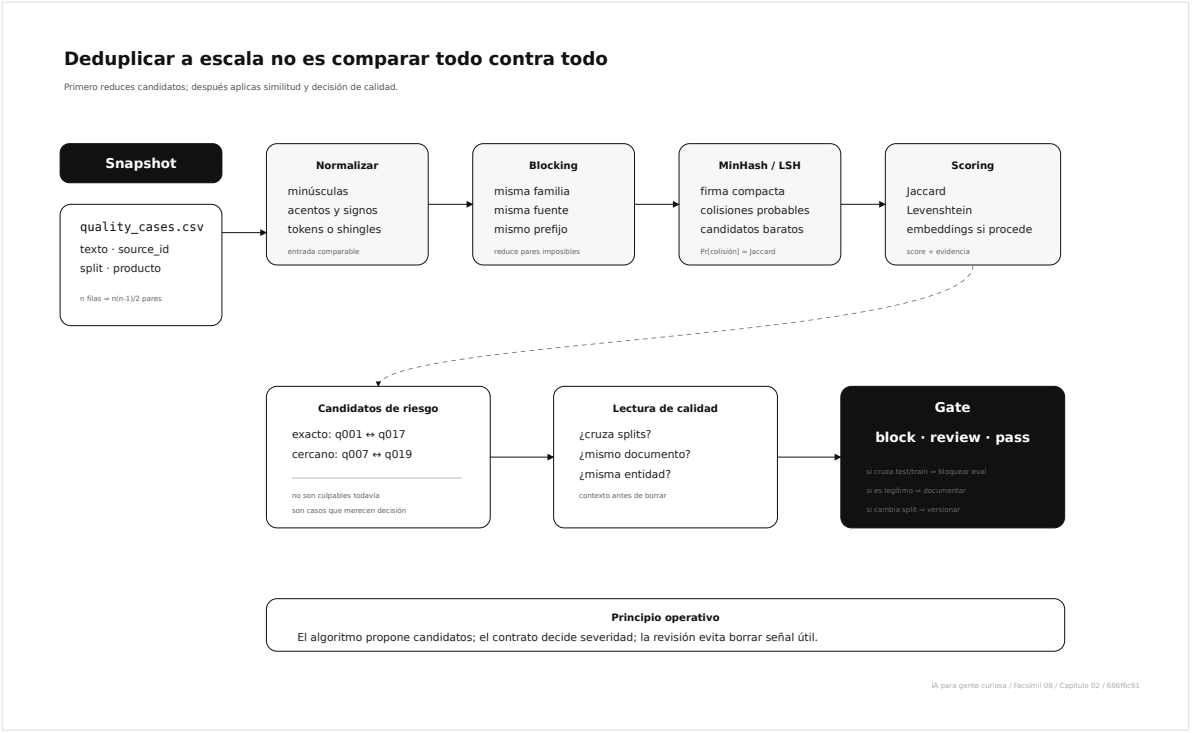
Broder formalizó una idea que sigue siendo central en deduplicación documental: estimar la semejanza de conjuntos mediante muestreo/fingerprints en lugar de comparar documentos completos.[13] En MinHash, si aplicamos una permutación aleatoria π y nos quedamos con el elemento mínimo de cada conjunto, se cumple:

$$\Pr[\min(\pi(A)) = \min(\pi(B))] = J(A, B)$$

SÍMBOLO	SIGNIFICADO	LECTURA DE INGENIERÍA
A, B	Conjuntos de tokens, shingles o fingerprints.	Dos textos o documentos normalizados.
π	Permutación aleatoria o función hash que la aproxima.	Una forma reproducible de reordenar elementos.
$\min(\pi(A))$	Valor mínimo de la firma para A.	Un componente de la firma MinHash.
$\Pr[\cdot]$	Probabilidad de colisión entre firmas.	Si colisionan mucho, probablemente se parecen.
$J(A, B)$	Similitud de Jaccard real.	La cantidad que queremos estimar sin comparar todo.

Esto no significa que MinHash “pruebe” que dos ejemplos son duplicados. Significa que ayuda a proponer candidatos con coste manejable. Después el gate debe mirar severidad: si el candidato cruza train y test, puede bloquear evaluación; si está dentro del mismo split, quizá

se revisa; si es un documento casi contenido en otro, puede afectar a RAG o a entrenamiento, no necesariamente a lo mismo.



> Para datasets grandes, deduplicar exige reducir candidatos antes de calcular similitudes caras. MinHash ayuda a escalar la búsqueda; el gate decide qué hacer con cada candidato.

Leakage entre splits

Leakage no es solo duplicado textual. Es cualquier información que permite al sistema obtener ventaja en evaluación porque ya vio directa o indirectamente la respuesta.

En el cuaderno del facsímil usamos un procedimiento auditable: comparar ejemplos de test contra train con coincidencia exacta y Jaccard, revisar los candidatos y bloquear si cruzan splits por encima del umbral escrito en el contrato.

EVIDENCIA DE LEAKAGE	CÓMO SE DETECTA	QUÉ SIGNIFICA
Texto exacto cruzado	Coincidencia después de normalizar.	Test contiene algo ya visto por train.
Texto muy parecido	Jaccard alto entre tokens.	El caso puede estar midiendo memoria de plantilla.
Misma entidad	`student_id`, `document_id` o `source_id` cruzan splits.	El sistema puede reconocer contexto repetido.
Tiempo mal ordenado	Train contiene información posterior a test.	El modelo aprende futuro.

En el próximo capítulo entraremos más a fondo en splits y leakage. Aquí basta con una regla operativa: si algo de test está demasiado cerca de train, no uses ese test para decidir si el sistema generaliza.

La palabra “cerca” es deliberadamente incómoda. A veces la cercanía es exacta: misma fila, mismo texto, mismo `case_id`. Otras veces es semántica: dos consultas redactadas distinto pero con la misma solución, dos chunks del mismo documento, dos tickets de la misma persona o una feature que resume el futuro. Un buen gate no se queda en igualdad de strings. Produce candidatos, muestra evidencia y obliga a una revisión cuando el riesgo de contaminación afecta a la evaluación.

En LLMs aparece una variante especialmente incómoda: contaminación de benchmarks. Un modelo puede haber visto durante entrenamiento parte de un benchmark, sus soluciones o textos muy próximos. Sainz y otros insisten en que la contaminación debe medirse por benchmark y no suponerse resuelta de forma genérica.[14] Para un ingeniero de datos, la traducción es concreta: guarda hashes, snapshots, fechas de corte, fuentes de benchmark y reglas de exclusión. Si no puedes explicar qué datos no pudieron entrar en entrenamiento, tu evaluación tiene un agujero.

Distribución de etiquetas

Una etiqueta no es solo un texto. Es una decisión que influye en el aprendizaje o en la evaluación. Por eso conviene mirar proporciones. Lo que escribimos abajo es la frecuencia relativa muestral de una clase: una estimación empírica de la distribución de etiquetas.[15]

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
y	Etiqueta.	`answer`, `ask_more`, `escalate`.
k	Clase concreta.	`ask_more`.
n_k	Número de ejemplos de esa clase.	4 casos.
n	Número total de ejemplos.	19 casos.

Si casi todo es `answer`, el sistema puede aprender a contestar siempre. Si casi no hay `escalate`, puede parecer correcto hasta que aparece un caso crítico. En clasificación y evaluación, mirar la distribución por split es obligatorio.

En un proyecto real, esta revisión debería hacerse antes de mirar el modelo. Si una clase crítica aparece dos veces en test, cualquier porcentaje será frágil. Si una clase domina train, un baseline perezoso puede parecer competente. Y si la distribución cambia entre train, validation y test, quizá el problema no sea el algoritmo sino la muestra. La distribución de etiquetas no decide sola, pero sí te dice si la evaluación tiene suelo suficiente para sostener una conclusión.

Otra forma académica de medir concentración de etiquetas es la entropía de Shannon.[16] Si Y es la variable etiqueta:

$$H(Y) = -\sum_{k=1}^{K} p_k \log p_k$$

SÍMBOLO	SIGNIFICADO	LECTURA PRÁCTICA
K	Número de clases posibles.	`answer`, `ask_more`, `escalate`.
p_k	Proporción de la clase k.	Frecuencia relativa de `escalate`.
$H(Y)$	Entropía de la distribución de etiquetas.	Baja si casi todo cae en una clase.

La entropía no dice si las etiquetas son correctas. Solo mide concentración. Si $H(Y)$ cae mucho entre versiones, puede haber cambio real de demanda, sesgo de muestreo, error de pipeline o limpieza que borró casos difíciles. En ingeniería de datos no se usa como veredicto; se usa como señal para preguntar mejor.

Para comparar la distribución global con la de cada split podemos usar distancia total variation, una medida estándar de distancia entre distribuciones de probabilidad.[17]

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
P_i	Proporción global de la clase i.	Proporción global de `answer`.
Q_i	Proporción de la clase i en un split.	Proporción de `answer` en test.
D_{TV}	Distancia entre distribuciones.	0.172932 para test en el cuaderno.

scikit-learn permite hacer splits estratificados con `train_test_split(..., stratify=y)` cuando esa estrategia encaja con el problema.[18] Estratificar no arregla etiquetas malas, pero ayuda a que cada partición conserve una mezcla parecida de clases.

Calidad de etiquetas y acuerdo entre anotadores

Las etiquetas son datos, no dogmas. En el cuaderno aparecen tres señales de revisión: desacuerdo entre anotadores, diferencia con una referencia didáctica y baja confianza de un modelo auxiliar. En un proyecto real no siempre tendrás `expected_label`, pero sí puedes tener doble anotación, reglas de revisión, muestras auditadas o modelos que prioricen casos dudosos.

El acuerdo observado se calcula así:

$$p_o = \frac{\sum_{i=1}^{n} 1[a_i = b_i]}{n}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a_i	Etiqueta del anotador A para el ejemplo i.	`answer`.
b_i	Etiqueta del anotador B para el ejemplo i.	`ask_more`.
p_o	Proporción de acuerdo observado.	0.789474 en el cuaderno.

Pero parte del acuerdo puede ocurrir por azar, especialmente si una clase domina. Kappa de Cohen corrige ese efecto:[19]

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
p_o	Acuerdo observado.	0.789474.
p_e	Acuerdo esperado por azar según distribuciones marginales.	0.518006.
κ	Acuerdo corregido por azar.	0.563218.

En el cuaderno, κ queda por debajo del umbral `0.6`, así que no bloquea por sí solo, pero exige revisión. Esta distinción importa: una etiqueta dudosa no siempre se borra; se revisa con política.

Más de dos anotadores: Fleiss y Krippendorff

Cohen encaja cuando hay dos anotadores. En proyectos de datos para IA no siempre es así. Puedes tener tres revisores por caso, un adjudicador, revisiones parciales o anotadores distintos según el producto. En ese contexto, usar la fórmula de dos anotadores por comodidad puede dar una falsa sensación de rigor.

Para etiquetas nominales con varios anotadores por ítem, una referencia clásica es kappa de Fleiss.[20] Su forma agregada mantiene la misma intuición que Cohen: compara acuerdo observado contra acuerdo esperado por azar.

$$\kappa_F = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e}$$

Donde, si cada ejemplo recibe m anotaciones y n_{ij} es cuántos anotadores asignaron la clase j al ejemplo i:

$$P_i = \frac{1}{m(m-1)} \sum_{j=1}^{k} n_{ij}(n_{ij} - 1)$$

$$\bar{P} = \frac{1}{N} \sum_{i=1}^{N} P_i$$

$$\bar{P}_e = \sum_{j=1}^{k} p_j^2$$

SÍMBOLO	SIGNIFICADO
N	Número de ejemplos anotados.
m	Número de anotaciones por ejemplo.
k	Número de clases posibles.
n_{ij}	Anotadores que eligieron la clase j para el ejemplo i.
P_i	Acuerdo entre anotadores en el ejemplo i.
$\bar{P}$	Acuerdo medio observado.
$\bar{P}_e$	Acuerdo esperado por azar según prevalencia de clases.

Krippendorff propuso otra familia de coeficientes especialmente útil cuando hay anotaciones incompletas, distintos niveles de medida o datos de contenido.[21] Su intuición se resume así:

$$\alpha = 1 - \frac{D_o}{D_e}$$

SÍMBOLO	SIGNIFICADO
D_o	Desacuerdo observado.
D_e	Desacuerdo esperado por azar.
α	Fiabilidad corregida por azar.

En este capítulo no vamos a calcular Fleiss ni Krippendorff en el cuaderno, porque el ejercicio trae dos anotadores para que Cohen sea transparente. Pero sí conviene que un ingeniero de datos lo tenga en la cabeza: si tu proceso real tiene más de dos personas, anotaciones incompletas o escalas ordinales, la métrica de acuerdo también forma parte del contrato. No vale decir “hay consenso” sin definir cómo se midió.

Medir la cola de revisión: precisión y exhaustividad

Cuando una herramienta marca posibles errores de etiqueta, no conviene creerla como si fuera una autoridad. Conviene evaluarla como evaluaríamos un detector. La pregunta no es solo “¿ha encontrado casos raros?”, sino “¿cuántos de los casos que marcó eran errores reales?” y “¿cuántos errores reales se le escaparon?”. En recuperación de información y clasificación, estas preguntas se expresan con precisión y exhaustividad, normalmente llamada *recall* en documentación técnica.[22]

$$precision = \frac{TP}{TP + FP}$$

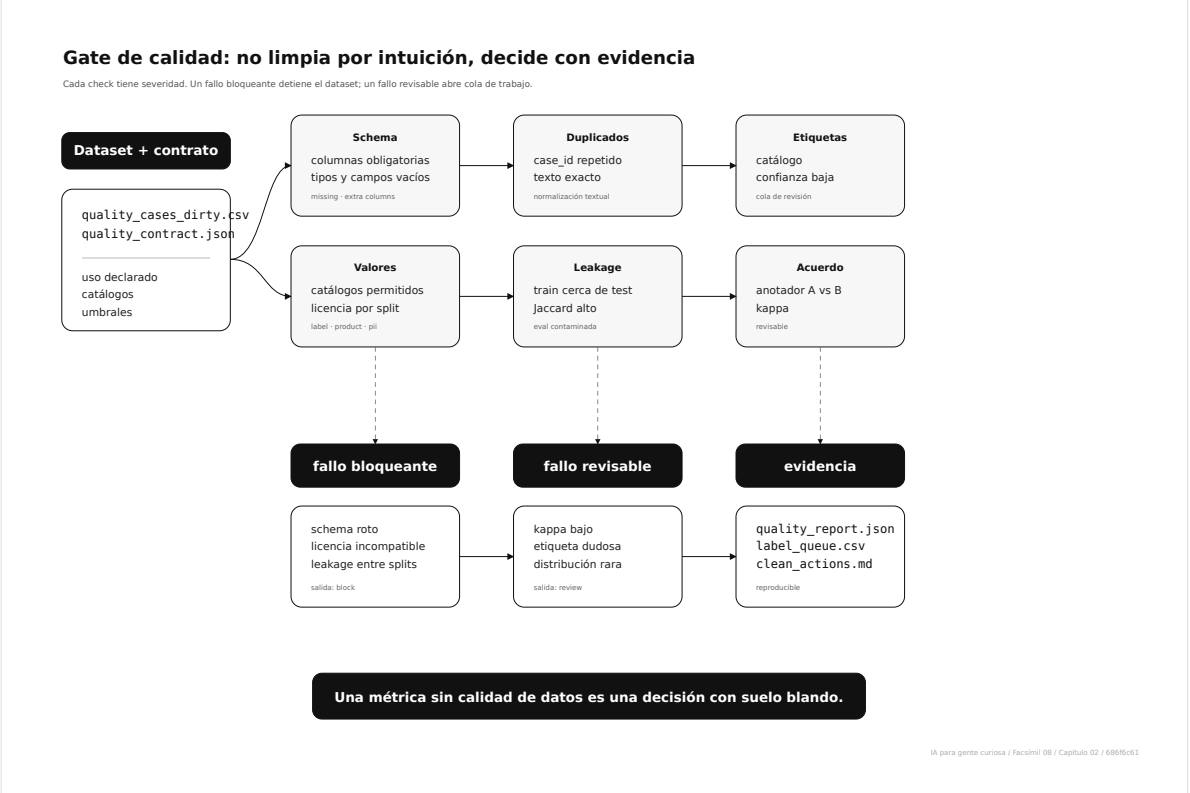
$$recall = \frac{TP}{TP + FN}$$

SÍMBOLO	SIGNIFICADO EN REVISIÓN DE ETIQUETAS
TP	Casos marcados como dudosos que, tras revisión humana, eran errores reales.
FP	Casos marcados como dudosos que estaban bien etiquetados.
FN	Casos que eran errores reales, pero el detector no propuso revisar.

Esto baja la herramienta a tierra. Si la cola tiene mucha precisión pero poco recall, revisas pocos casos y casi todos aportan valor, pero se te escapan errores. Si tiene mucho recall y poca precisión, atrapas casi todo, pero saturas al equipo con falsos positivos. En un dataset pequeño de evaluación, quizá prefieres recall alto porque cada etiqueta pesa mucho. En entrenamiento masivo, quizá priorizas precisión para no convertir la revisión en una fábrica imposible.

El cuaderno no pretende resolver automáticamente este equilibrio. Lo enseña de forma controlada: `label_review_queue.csv` no es una lista de culpables, es una bandeja de entrada para aplicar la política de anotación, registrar decisiones y medir si la heurística de revisión está ayudando o solo generando ruido.

Anatomía de un gate de calidad



Un gate de calidad separa fallos bloqueantes, revisiones necesarias y evidencia generada. No sustituye el criterio humano: lo organiza.

Severidad: cuándo bloquear, cuándo revisar y cuándo aceptar deuda

Un gate útil no trata todos los fallos igual. Si una etiqueta dudosa y una licencia incompatible aparecen en la misma lista sin severidad, el equipo pierde criterio. La severidad existe para responder a una pregunta muy concreta: **¿qué daño produce avanzar con este fallo?**

En ingeniería de IA conviene separar tres niveles. Un fallo bloqueante invalida el uso automático del dataset. Un fallo revisable no permite publicar una decisión todavía, pero sí permite abrir trabajo humano. Una deuda aceptada es un problema conocido que no afecta al uso actual o que tiene un plan explícito de corrección.

NIVEL	PREGUNTA DE DECISIÓN	EJEMPLO	ACCIÓN
`block`	¿Puede invalidar entrenamiento, evaluación, permiso o seguridad del uso?	Leakage entre `train` y `test`; licencia incompatible; etiqueta fuera de catálogo.	Detener uso automatizado.
`review`	¿Necesita criterio humano antes de confiar en la decisión?	Kappa bajo; baja confianza de etiqueta; distribución rara.	Abrir cola de revisión.
`accepted_debt`	¿Se puede vivir con esto durante una ventana concreta?	Missing en un campo no usado por este modelo.	Documentar owner, fecha y plan.

Esta tabla evita dos extremos. El primero es bloquearlo todo y convertir calidad en un cuello de botella imposible. El segundo es dejar pasar todo porque “ya lo arreglaremos”. La severidad obliga a escribir el riesgo y la acción. Si no puedes explicar por qué algo es `block` o `review`, quizá la regla todavía no está bien formulada.

La matriz también debe depender del uso. Un `source_id` vacío puede ser revisable para exploración local, pero bloqueante para evaluación trazable. Una etiqueta dudosa puede ser tolerable en entrenamiento con millones de ejemplos, pero bloqueante en un test pequeño que decide entre dos modelos. La severidad no vive aislada: vive con el contrato.

SLI, SLO y presupuesto de error de datos

En operación ya vimos que un SLI es un indicador medible, un SLO es un objetivo y el presupuesto de error dice cuánto margen queda antes de actuar. Esa idea encaja muy bien con calidad de datos.

Un **SLI de datos** mide una propiedad concreta del dataset. Un **SLO de datos** fija el umbral aceptable. Un **presupuesto de error de datos** define qué ocurre cuando nos acercamos al límite o lo superamos. No es una frase de marketing: es una herramienta para que el equipo sepa cuándo parar.

SLI DE DATOS	QUÉ MIDE	SLO POSIBLE	ACCIÓN SI FALLA
`critical_missing_rate`	Missing en columnas críticas.	`<= 0.5%`	Bloquear snapshot si afecta linaje o etiqueta.
`cross_split_duplicates`	Duplicados entre splits.	`0`	Bloquear evaluación.
`invalid_catalog_values`	Valores fuera de catálogo.	`0`	Bloquear hasta migrar contrato o datos.
`license_mismatches`	Uso incompatible con licencia.	`0`	Bloquear uso automatizado.
`annotator_kappa`	Acuerdo corregido por azar.	`>= 0.6`	Abrir revisión de política de anotación.
`label_review_queue_size`	Casos pendientes de revisar.	`<= 2`	No cerrar release si supera el umbral.

En el cuaderno del facsímil, estos objetivos aparecen en `contracts/quality_slos.json`. Es importante verlo así: no es una ruta para buscar dentro del repositorio, sino un archivo que puedes abrir, modificar y versionar como harías en un proyecto real.

El presupuesto de error evita discusiones vagas. Por ejemplo: si el SLO exige cero duplicados entre train y test, un solo duplicado consume todo el presupuesto y bloquea. Si el SLO permite hasta dos casos en cola de revisión, un tercero no significa que el dataset sea inútil, pero sí que la release no debería cerrarse sin mirar esos casos.

Ciclo de vida de una incidencia de datos

Detectar un fallo es el principio, no el final. Una incidencia de datos debería tener un ciclo de vida tan claro como una incidencia de software:

1. Detectar: el gate marca un check fallido.
2. Clasificar: se decide severidad, owner y uso afectado.
3. Diagnosticar: se busca causa raíz.
4. Corregir: se cambia dato, contrato, pipeline o política.
5. Reejecutar: se vuelve a correr el gate.
6. Versionar: se guarda snapshot, reporte y decisión.
7. Aprender: se añade un check para que no vuelva a pasar igual.

La causa raíz importa porque no todos los fallos se arreglan tocando el CSV. Si aparece una etiqueta `resolve`, puede ser un error de anotación, pero también una señal de que el dominio necesita una nueva clase. Si aparece `admisiones` en `product`, puede ser un campo mal escrito, o puede ser que el producto haya crecido y el contrato esté viejo. Si baja kappa, quizá los anotadores fallaron, pero quizá la política era ambigua.

SÍNTOMA	CAUSA RAÍZ POSIBLE	ARREGLO POBRE	ARREGLO DE INGENIERÍA
Etiqueta nueva	Catálogo real cambió.	Reemplazarla por la clase más parecida.	Versionar contrato y migrar evaluación.
Duplicado entre splits	Split creado después de deduplicar mal.	Borrar una fila sin registro.	Rehacer split desde snapshot limpio.
Kappa bajo	Política de anotación ambigua.	Pedir “más cuidado”.	Añadir casos frontera y regla de desempate.
Licencia incompatible	Export mezcló permisos.	Ignorar en prototipo.	Separar usos permitidos por split.
Missing en linaje	Pipeline perdió metadatos.	Rellenar a mano.	Corregir ingestión y reejecutar.

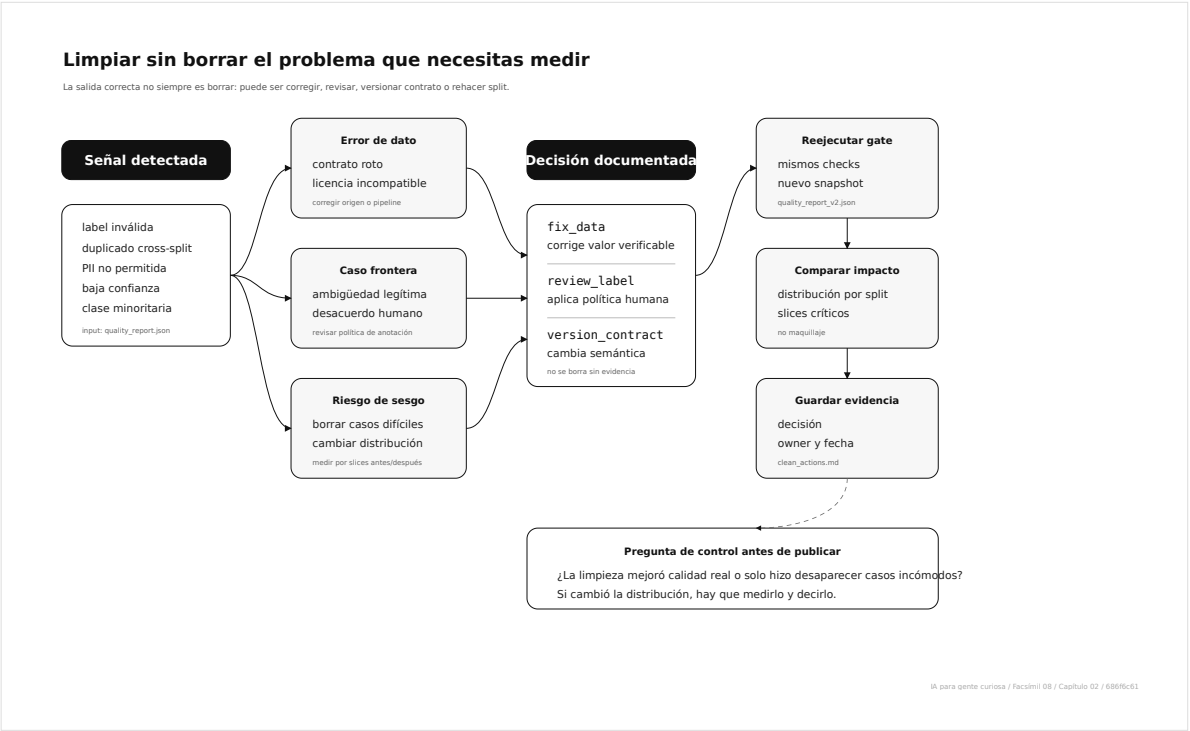
Esta forma de pensar es muy de ingeniería: no arreglar solo el síntoma, sino el mecanismo que permitió que apareciera.

Limpiar sin destruir señal

Una limpieza mala suele tener buena intención. Alguien mira el dataset, ve casos difíciles y piensa: “si quito esto, la métrica sube”. A veces sube. El problema es que quizá ya no mide el problema real. En IA, limpiar no puede significar borrar todo lo que molesta. Tiene que significar separar errores de datos, casos frontera legítimos y ejemplos difíciles que el sistema necesita aprender o evaluar.

La diferencia es importante. Una fila con `label=resolve` puede ser un error de catálogo. Una pregunta ambigua sobre becas puede ser un caso legítimo de `ask_more`. Un pago duplicado puede ser una incidencia real que debe escalarse, no un outlier que conviene eliminar. Si el equipo borra sin política, convierte la calidad en maquillaje.

CASO	QUÉ NO HACER	QUÉ HACER
Etiqueta inválida	Mapearla a mano a la clase más frecuente.	Abrir incidencia: ¿error de anotación o nueva clase?
Duplicado entre splits	Borrar el que “moleste menos”.	Rehacer split desde snapshot deduplicado y guardar manifiesto.
Baja confianza	Eliminar todos los casos de baja confianza.	Revisar muestra, medir precisión/recall de la cola y ajustar política.
Clase minoritaria	Balancear borrando clases frecuentes sin criterio.	Diseñar muestreo, pesos o evaluación por slices.
PII alta	Enmascarar sin registrar transformación.	Aplicar política de privacidad, dejar linaje y revalidar.



Limpiar bien no consiste en borrar filas molestas. Consiste en clasificar el problema, aplicar una acción trazable, reejecutar el gate y comprobar que no hemos sesgado el dataset.

Schema evolution: cuando el dataset cambia sin pedir permiso

Los datasets vivos cambian. Aparecen columnas, se renombran campos, se añaden etiquetas, se deprecan productos y cambian tipos. Ese cambio no es malo. Lo peligroso es que ocurra sin versión, sin migración y sin compatibilidad.

Schema evolution significa gestionar cambios de forma controlada. En un dataset de IA, no basta con que el código lea la nueva columna. Hay que saber si el significado cambió. Una columna `label` puede mantener el mismo nombre y cambiar por completo si el equipo añade una política nueva. Un valor `resolve` puede ser un error hoy y una clase legítima mañana.

CAMBIO	RIESGO	ESTRATEGIA
Nueva columna opcional	Los consumidores antiguos la ignoran.	Añadirla como `optional` y documentar fecha.
Columna obligatoria nueva	Pipelines antiguos fallan.	Crear versión `v2` del contrato y periodo de convivencia.
Nuevo valor de catálogo	Métricas antiguas no comparan igual.	Versionar label set y rehacer splits/evals.
Cambio de tipo	Validación y transformación pueden romper.	Migración explícita con tests de datos.
Cambio de significado	El peor caso: parece compatible, pero no lo es.	Nota de contrato, ejemplos y revisión humana.

Una regla útil: si un cambio altera una decisión, no es un detalle técnico. Es una nueva versión del contrato.

Política de anotación: la etiqueta también necesita contrato

Una etiqueta como `answer` parece sencilla hasta que aparecen casos frontera. ¿Qué hacemos si falta un dato menor? ¿Y si el usuario mezcla beca y matrícula? ¿Y si el asistente podría responder, pero con riesgo de orientar mal? Sin política, cada anotador inventa su criterio.

El cuaderno del facsímil incluye una política mínima en `contracts/annotation_policy.md`. Ese archivo es deliberadamente pequeño, pero tiene una función seria: convertir desacuerdos humanos en reglas revisables, no en conversaciones de pasillo.

Una política de anotación profesional debería incluir:

PIEZA	POR QUÉ IMPORTA
Definición de cada etiqueta	Evita que dos personas usen palabras iguales con criterios distintos.
Ejemplos positivos	Muestran cuándo sí aplica la etiqueta.
Ejemplos negativos	Muestran cuándo no aplica aunque se parezca.
Casos frontera	Reducen desacuerdo donde el dominio es ambiguo.
Regla de desempate	Permite cerrar conflictos sin improvisar.
Versión de política	Hace trazable qué criterio produjo cada etiqueta.

Esto conecta directamente con kappa. Si el acuerdo es bajo, no siempre significa que los anotadores sean malos. Puede significar que la política no explica los casos difíciles. En ese caso, el siguiente paso no es reetiquetar a ciegas: es mejorar la política y después revisar.

Calidad por segmentos

Una media global puede esconder un problema local. El dataset puede tener buen missing global, pero fallar en `becas`; buen kappa global, pero mucho desacuerdo en `chat`; buen reparto de etiquetas, pero ningún caso crítico en `test`.

Por eso la calidad debería mirarse por segmentos:

SEGMENTO	PREGUNTA
`product`	¿Todos los temas tienen cobertura y etiquetas coherentes?
`channel`	¿Chat, email y portal tienen estilos comparables?
`split`	¿Train, validation y test mantienen distribuciones razonables?
`pii_risk`	¿Los casos sensibles tienen controles suficientes?
`language`	¿La calidad se mantiene si hay varios idiomas?
`criticality`	¿Los casos de alto impacto están representados y revisados?

Esta idea preparará el capítulo de slices. La intuición es sencilla: un dataset no se rompe siempre “en general”. Muchas veces se rompe en un rincón. La ingeniería consiste en encontrar ese rincón antes de que lo encuentre producción.

Leakage temporal

Hasta ahora hemos hablado de leakage entre splits. Hay otro tipo especialmente traicionero: leakage temporal. Ocurre cuando el dataset usa información que no existía en el momento en que el sistema habría tenido que decidir.

Ejemplos:

CASO	LEAKAGE TEMPORAL
Predicción de urgencia	Usar `days_to_resolution`, que solo se conoce después de cerrar el caso.
RAG académico	Evaluar con un documento actualizado después de la consulta original.
Feature de pagos	Usar un estado de pago corregido días después del evento.
Etiqueta de soporte	Etiquetar con información de resolución posterior y usarla como entrada.

La regla de oro es: para cada feature o documento, pregunta "¿esto estaba disponible en ese momento?". Si la respuesta es no, quizá sirve para análisis histórico, pero no para entrenar o evaluar una decisión que pretende simular producción.

En el día a día

En un equipo de IA, la calidad de datos debería estar cerca de CI. Igual que un cambio de código no debería desplegarse si rompe tests, un snapshot de datos no debería llegar a entrenamiento o evaluación si rompe checks críticos.

La práctica madura tiene cuatro capas:

CAPA	PREGUNTA	ARTEFACTO
Contrato	¿Qué prometen los datos?	JSON, YAML, ODCS, schema programático.
Validación	¿Cumplen lo prometido?	Reporte de checks y anomalías.
Revisión	¿Qué casos requieren criterio?	Cola de etiquetas, duplicados, incidencias.
Decisión	¿Qué hacemos con el snapshot?	`pass`, `review`, `block` y plan de limpieza.

La parte difícil no es escribir `if row["label"] not in allowed_labels`. Eso lo puede hacer cualquiera. La parte difícil es decidir qué fallos bloquean, cuáles abren revisión y cuáles se aceptan temporalmente con una nota. Ahí aparece la ingeniería: convertir incertidumbre en reglas operativas que un equipo pueda sostener.

En CI/CD, el patrón mínimo sería:

```
python3 ops/data_quality_gate.py --write
python3 ops/ci_assert_gate.py
```

Si `release_gate.json` contiene `block`, el segundo comando termina con error y el pipeline se detiene. Si el equipo permite releases en `review` para entornos manuales, podría ejecutarlo con:

```
python3 ops/ci_assert_gate.py --allow-review
```

La clave es que el pipeline no decide por intuición. Lee un artefacto. Ese artefacto queda guardado y puede revisarse después.

Herramientas por capa

Las herramientas importan, pero no por el logo. Importan porque obligan a convertir dudas en artefactos: expectations, schemas, reportes, anomalías, colas de revisión o alertas. Si una herramienta no deja evidencia revisable, en este capítulo no nos sirve demasiado.

CAPA	HERRAMIENTAS	QUÉ APORTAN	QUÉ NO DELEGARÍA
Validación declarativa	Great Expectations, Pandera, Deequ, Soda Core	Checks sobre columnas, nulos, rangos, catálogos, reglas de tabla y contratos.	Decidir severidad de negocio o si un fallo invalida una evaluación.
Validación ML pipeline	TensorFlow Data Validation	Schema, estadísticas, anomalías y comparación entre datasets dentro de pipelines TFX.	Entender por sí sola si una etiqueta es pedagógicamente correcta.
Datos y etiquetas	Cleanlab, Snorkel	Detección/priorización de errores de etiqueta y creación de señales de supervisión débil.	Convertir una señal probabilística en verdad sin revisión.
Monitorización	Evidently, Soda, observabilidad de datos	Drift, estadísticas, calidad recurrente y comparación contra referencia.	Sustituir un contrato de datos escrito.
CI/CD de datos	Scripts propios, dbt tests, Airflow/Dagster assets	Ejecutar gates, guardar outputs y bloquear snapshots.	Arreglar automáticamente un fallo sin política.

Great Expectations trabaja con expectations sobre datos; Pandera permite expresar schemas de DataFrames en código; Deequ acerca la idea de tests de datos a Spark; Soda Core permite definir checks y contratos; TFDV perfila datasets y detecta anomalías; Cleanlab ayuda a encontrar posibles problemas de etiqueta; Snorkel estructura supervisión débil; Evidently se usa para evaluar y monitorizar cambios de datos y modelos.[23242526272829]

La decisión práctica sería esta: usa una herramienta declarativa para reglas obvias, una herramienta de perfilado para cambios de distribución, una herramienta de etiqueta para priorizar revisión y un gate propio para convertir todo eso en `pass`, `review` o `block`. El gate propio no tiene que ser grande. Tiene que ser explícito, versionado y entendible por el equipo.

Herramientas externas de mercado: cuándo tienen sentido

Fecha de corte de esta sección: 21 de junio de 2026. El mercado de data observability cambia deprisa, así que lo importante no es memorizar proveedores. Lo importante es entender qué problema operativo resuelve cada familia y qué evidencia debería producir.

Una herramienta externa tiene sentido cuando el problema ya no cabe en un script local: muchas tablas, muchos owners, varios warehouses, alertas recurrentes, incidentes con impacto, linaje que cruza equipos, necesidad de priorizar por negocio o auditoría de cambios entre entornos. Si solo tienes un CSV y un contrato pequeño, el cuaderno del facsímil es mejor profesor que una plataforma. Si tienes cien pipelines críticos, necesitas otra cosa.

HERRAMIENTA	DÓNDE ENCAJA	QUÉ DEBERÍAS PEDIRLE COMO EVIDENCIA	RIESGO SI LA USAS MAL
Monte Carlo	Observabilidad de datos y AI/data pipelines: monitores, alertas, impacto y resolución.	Incidente, tabla afectada, patrón anómalo, lineage/impacto y owner notificado.	Creer que una alerta sustituye al contrato semántico del dataset.
Bigeye	Observabilidad empresarial con reglas, anomalías, lineage, reconciliación e incidentes.	Métrica monitorizada, umbral o anomalía, histórico, impacto y flujo de resolución.	Comprar visibilidad sin definir qué bloquea una release de IA.
Anomalo	Monitorización automática y detección de anomalías en warehouses grandes.	Dato tardío, faltante o anómalo, explicación de root cause y prioridad.	Confundir anomalía estadística con error de negocio.
Datafold	Data diff: comparación de tablas, columnas, filas o migraciones.	Diferencias exactas entre fuente y destino, filas afectadas y columnas cambiadas.	Usarlo solo para migraciones y olvidar que un cambio semántico puede no verse como diff obvio.
Soda	Checks, contratos, observabilidad y workflow de calidad.	Check fallido, contrato afectado, owner y evolución del problema.	Llenar el sistema de checks sin severidad ni decisión.
Great Expectations Cloud/Core	Expectations declarativas y validación reproducible.	Suite de expectations, resultado por expectation y Data Docs/reporte.	Tener expectativas técnicas sin conectar con uso de IA.
Cleanlab	Priorización de problemas de etiqueta o datos.	Ranking de casos sospechosos y señales de incertidumbre.	Tratar un score como verdad sin revisión humana.
Evidently	Evals, drift, calidad y monitorización de modelos/datos.	Reporte comparando referencia contra producción o batch actual.	Mirar drift global y no revisar slices críticos.

Monte Carlo se presenta como una plataforma de observabilidad de datos y AI/data pipelines que aprende patrones y alerta sobre incidencias en warehouses, lakes, ETL y BI.[30] Bigeye describe una plataforma de data observability con lineage, anomalías, reglas, reconciliación e incident management.[31] Anomalo enfatiza monitorización automática con aprendizaje no supervisado para detectar datos tardíos, faltantes o anómalos a escala.[32] Datafold documenta `Data Diff` como comparación a nivel de valores entre tablas para detectar cambios críticos.[33]

La regla de compra para un ingeniero de datos sería sobria: antes de pagar una plataforma, escribe tres incidentes que quieres detectar, tres decisiones que quieres bloquear y tres evidencias que una persona revisora necesita leer. Si la herramienta no puede producir esas evidencias, no está resolviendo este capítulo; solo está poniendo una interfaz encima.

Por qué debería importarte

Si no revisas calidad, puedes entrenar con etiquetas rotas, evaluar con leakage, elegir un modelo por una métrica falsa o indexar documentos que no debían usarse. Todo eso produce sistemas que parecen funcionar hasta que llegan a producción o hasta que alguien intenta explicar una decisión.

Además, la calidad de datos es una de las formas más baratas de mejorar IA. Cambiar de modelo puede costar dinero, latencia y complejidad. Corregir una etiqueta mal puesta, separar bien splits o bloquear una licencia incompatible puede mejorar la decisión sin tocar arquitectura.

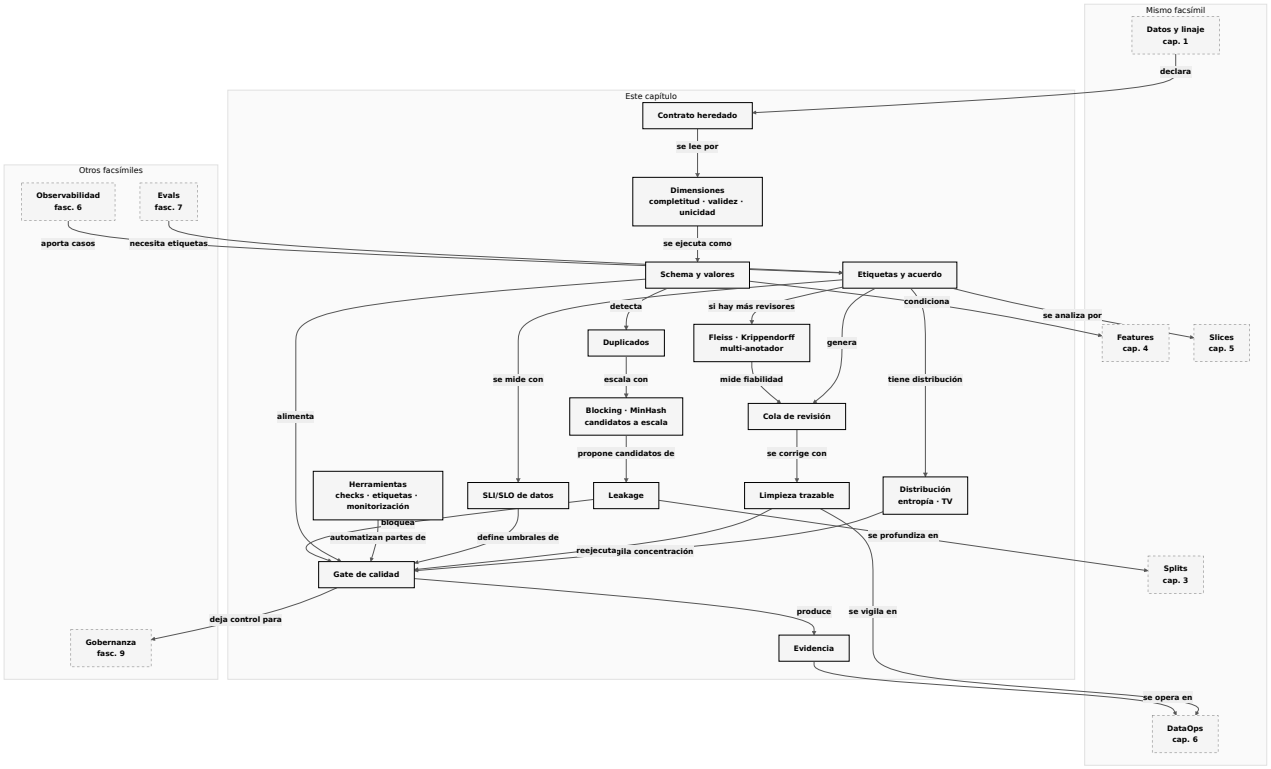
La calidad no compite con el modelado. Lo hace posible.

Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Mirar solo nulos	Es el check más fácil de programar.	Separar schema, catálogo, licencias, duplicados, leakage y etiquetas.
Borrar duplicados sin mirar splits	Parece una limpieza obvia.	Revisar si el duplicado contamina evaluación o representa un evento legítimo.
Tratar una etiqueta como verdad	La columna `label` da falsa seguridad.	Medir acuerdo, revisar baja confianza y documentar política de anotación.
Mezclar fallos bloqueantes y revisables	Todo se mete en una lista única.	Asignar severidad: `block` para lo que invalida uso, `review` para lo que pide criterio.
Celebrar un `pass` sin contexto	Suena a certificado universal.	Leerlo siempre junto a contrato, uso, fecha y versión del dataset.
Cambiar schema sin versión	El código sigue funcionando y parece compatible.	Versionar contrato si cambia una decisión o significado.
Mirar calidad solo global	Las medias esconden problemas locales.	Revisar por producto, canal, split, sensibilidad e idioma.
Borrar casos difíciles para subir la métrica	La limpieza se confunde con maquillaje.	Separar error, caso frontera y señal útil antes de eliminar.
Creer a la cola de revisión sin medirla	La herramienta suena objetiva.	Medir precisión/recall con una muestra revisada por personas.
Elegir herramienta antes de contrato	Es más cómodo instalar que pensar.	Definir uso, severidad y evidencia antes de automatizar.
Comparar todo contra todo	Funciona en una demo pequeña y se rompe a escala.	Usar blocking, fingerprints y candidatos antes del scoring caro.
Usar Cohen con cualquier proceso de anotación	Es la métrica que más suena.	Usar Cohen solo con dos anotadores; mirar Fleiss o Krippendorff cuando el proceso cambia.
Ignorar contaminación de benchmarks	El benchmark parece externo y limpio.	Guardar fechas de corte, hashes y reglas de exclusión.
Comprar observabilidad sin gate	La plataforma promete visibilidad.	Exigir evidencia, severidad y acción antes de integrarla en releases de IA.

Cómo encaja todo

El capítulo anterior nos dio contrato y linaje. Este capítulo ejecuta ese contrato. El siguiente capítulo se centrará en splits y leakage con más profundidad. Más adelante veremos features, slices, drift y análisis aplicado. La calidad de datos es el puente entre “tengo datos” y “puedo tomar una decisión con ellos”.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Calidad de datos	Grado en que un dataset cumple el contrato necesario para una decisión concreta.
Schema	Forma esperada de los datos: columnas, tipos, catálogos y reglas.
Expectation	Regla verificable sobre el dataset.
Gate	Decisión automatizada de `pass`, `review` o `block`.
Duplicado exacto	Registro que repite clave o texto normalizado.
Duplicado cercano	Registro muy parecido según una medida de similitud.
Jaccard	Similitud entre conjuntos basada en intersección y unión.
Leakage	Información que contamina entrenamiento, evaluación o tiempo de decisión.
Etiqueta	Clase o salida esperada asociada a un ejemplo.
Ruido de etiqueta	Sospecha de que la etiqueta registrada no sigue la política.
Kappa	Acuerdo entre dos anotadores corregido por azar.
Cola de revisión	Lista priorizada de casos que necesitan criterio humano.
SLI de datos	Indicador medible de salud del dataset.
SLO de datos	Objetivo mínimo aceptable para un indicador de datos.
Presupuesto de error de datos	Margen de fallos tolerables antes de bloquear o revisar.
Schema evolution	Gestión versionada de cambios en columnas, tipos y catálogos.
Leakage temporal	Uso de información que no existía en el momento de decisión.
Dimensión de calidad	Aspecto evaluable de la calidad: completitud, validez, consistencia, unicidad, actualidad o trazabilidad.
Validez	Cumplimiento de tipos, formatos, rangos y catálogos permitidos.
Consistencia	Ausencia de contradicciones entre campos, tablas, fuentes o permisos.

TÉRMINO	DEFINICIÓN BREVE
Unicidad	Garantía de que una entidad o ejemplo no aparece repetido cuando el contrato lo prohíbe.
Distancia de edición	Medida de cuántos cambios hacen falta para convertir una cadena en otra.
Precisión de revisión	Qué proporción de casos marcados como sospechosos eran problemas reales.
Recall de revisión	Qué proporción de problemas reales logró detectar la cola de revisión.
Observabilidad de datos	Capacidad de detectar, explicar y trazar cambios de calidad, schema, volumen o distribución.
Data diff	Comparación de datos entre versiones, tablas o sistemas para detectar cambios de filas, columnas y valores.
Data observability	Monitorización operativa de frescura, volumen, schema, distribución, lineage, anomalías e impacto.
Blocking key	Clave barata que reduce candidatos antes de calcular similitudes más caras.
MinHash	Técnica probabilística para estimar Jaccard con firmas compactas.
LSH	Familia de técnicas para encontrar candidatos similares sin comparar todo contra todo.
Entropía de etiquetas	Medida de concentración o diversidad de la distribución de clases.
Kappa de Fleiss	Acuerdo nominal corregido por azar para más de dos anotadores.
Alfa de Krippendorff	Fiabilidad corregida por azar útil con distintos tipos de anotación y datos incompletos.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Por qué calidad de datos no significa solo “sin nulos”?
2. ¿Qué diferencia hay entre schema y expectation?
3. ¿Por qué un valor fuera de catálogo puede bloquear un dataset?
4. ¿Qué diferencia hay entre duplicado exacto y duplicado cercano?
5. ¿Cómo se calcula Jaccard y qué limitación tiene?
6. ¿Por qué un duplicado entre train y test puede contaminar una evaluación?

7. ¿Qué significa que una etiqueta vaya a cola de revisión?
8. ¿Por qué kappa corrige el acuerdo observado?
9. ¿Qué diferencia hay entre `block`, `review` y `pass`?
10. ¿Qué artefacto usarías para justificar una decisión de calidad ante otra persona?
11. ¿Qué SLI y SLO definirías para duplicados entre splits?
12. ¿Cuándo un cambio de schema exige nueva versión del contrato?
13. ¿Qué problema resuelve una política de anotación?
14. ¿Qué es leakage temporal y por qué no lo detecta siempre un deduplicador?
15. ¿Cómo conectarías el gate de datos a CI/CD?
16. ¿Qué diferencia hay entre completitud, validez, consistencia y unicidad?
17. ¿Cuándo usarías Jaccard y cuándo distancia de edición?
18. ¿Por qué una cola de revisión necesita precisión y recall?
19. ¿Cómo limpiarías un dataset sin destruir clases minoritarias o casos frontera?
20. ¿Qué herramienta usarías para checks declarativos y cuál para priorizar errores de etiqueta?
21. ¿Por qué comparar todos los pares de un dataset grande no escala?
22. ¿Qué relación hay entre MinHash y Jaccard?
23. ¿Cuándo usarías Fleiss o Krippendorff en lugar de Cohen?
24. ¿Qué evidencia guardarías para defender que una evaluación no está contaminada?
25. ¿Qué parte de tu proceso de calidad traducirías a una plataforma externa y cuál mantendrías como contrato propio?

En resumen

IDEA	QUÉ TE LLEVAS
Calidad es calidad para un uso.	No existe un dataset bueno en abstracto.
El schema es el principio.	La forma importa, pero no basta.
Los duplicados pueden contaminar.	Especialmente si cruzan splits.
Leakage rompe la evaluación.	El test deja de medir generalización.
Las etiquetas son datos revisables.	Necesitan política, acuerdo y cola de revisión.
Un gate organiza la decisión.	`block`, `review` y `pass` separan tipos de acción.
La práctica debe dejar evidencia.	Reporte, cola, duplicados y plan de limpieza.
Los SLOs hacen operativa la calidad.	Un indicador sin objetivo no guía una decisión.
El schema evoluciona.	Los cambios de significado necesitan versión y migración.
La calidad se mira por segmentos.	El fallo puede vivir en un producto, canal o split concreto.
CI/CD también sirve para datos.	El pipeline puede bloquear snapshots con evidencia reproducible.
Limpiar también puede sesgar.	Borrar casos difíciles puede mejorar una métrica y empeorar el sistema.
Las herramientas producen señales, no verdades.	Great Expectations, TFDV, Cleanlab o Evidently ayudan, pero la severidad sigue siendo una decisión de ingeniería.
La revisión se mide.	Precisión y recall permiten saber si una cola ayuda o solo genera ruido.
Deduplicar exige escala.	Blocking, MinHash y scoring separan candidatos baratos de decisiones caras.
El acuerdo depende del proceso.	Dos anotadores, muchos anotadores y anotaciones incompletas no se miden igual.
La contaminación moderna es documental.	En LLMs hay que controlar snapshots, benchmarks, hashes y fechas de corte.
Las herramientas externas no sustituyen criterio.	Monte Carlo, Bigeye, Anomalo, Datafold o Soda ayudan si producen evidencia accionable.

Para saber más

AWS Labs. (2026). Deequ: Unit Tests for Data. https://github.com/awslabs/deequ

Broder, A. Z. (1997). On the Resemblance and Containment of Documents. *Proceedings. Compression and Complexity of Sequences 1997*, 21-29. https://doi.org/10.1109/SEQUEN.1997.666900

Cleanlab. (2026). Cleanlab Documentation. https://docs.cleanlab.ai/

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

Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley.

Evidently AI. (2026). Evidently Documentation. https://docs.evidentlyai.com/docs/library/overview

Fleiss, J. L. (1971). Measuring Nominal Scale Agreement among Many Raters. *Psychological Bulletin*, 76(5), 378-382. https://doi.org/10.1037/h0031619

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

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.

Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8), 707-710.

Little, R. J. A. y Rubin, D. B. (2019). *Statistical Analysis with Missing Data* (3.ª ed.). Wiley. https://doi.org/10.1002/9781119482260

Krippendorff, K. (2004). Reliability in Content Analysis: Some Common Misconceptions and Recommendations. *Human Communication Research*, 30(3), 411-433. https://doi.org/10.1111/j.1468-2958.2004.tb00738.x

Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Northcutt, C. G., Athalye, A. y Mueller, J. (2021). Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks. *NeurIPS Datasets and Benchmarks*. https://arxiv.org/abs/2103.14749

Northcutt, C. G., Jiang, L. y Chuang, I. L. (2021). Confident Learning: Estimating Uncertainty in Dataset Labels. *Journal of Artificial Intelligence Research*, 70, 1373-1411. https://doi.org/10.1613/jair.1.12125

Pandera. (2026). Pandera Documentation. https://pandera.readthedocs.io/en/stable/

Ratner, A., Bach, S. H., Ehrenberg, H., Fries, J., Wu, S. y Ré, C. (2017). Snorkel: Rapid Training Data Creation with Weak Supervision. *PVLDB*, 11(3), 269-282. https://doi.org/10.14778/3157794.3157797

scikit-learn. (2026). train_test_split. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

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

Sainz, O., Campos, J. A., García-Ferrero, I., Etxaniz, J., de Lacalle, O. L. y Agirre, E. (2023). NLP Evaluation in Trouble: On the Need to Measure LLM Data Contamination for each Benchmark. *Findings of ACL: EMNLP 2023*. https://doi.org/10.18653/v1/2023.findings-emnlp.722

Soda. (2026). What is Soda? https://docs.soda.io/

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

TensorFlow. (2026). TensorFlow Data Validation Anomalies Reference. https://www.tensorflow.org/tfx/data_validation/anomalies

Wang, R. Y. y Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33. https://doi.org/10.1080/07421222.1996.11518099

Notas

1. Wang, R. Y. y Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33. https://doi.org/10.1080/07421222.1996.11518099 ↩

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

3. TensorFlow. (2026). *TensorFlow Data Validation Anomalies Reference*. https://www.tensorflow.org/tfx/data_validation/anomalies. Consultado el 6 de junio de 2026. ↩

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

5. Pandera. (2026). *Pandera Documentation*. https://pandera.readthedocs.io/en/stable/. Consultado el 6 de junio de 2026. ↩

6. AWS Labs. (2026). *Deequ: Unit Tests for Data*. https://github.com/awslabs/deequ. Consultado el 6 de junio de 2026. ↩

7. Northcutt, C. G., Jiang, L. y Chuang, I. L. (2021). Confident Learning: Estimating Uncertainty in Dataset Labels. *Journal of Artificial Intelligence Research*, 70, 1373-1411. https://doi.org/10.1613/jair.1.12125 ↩

8. Northcutt, C. G., Athalye, A. y Mueller, J. (2021). Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks. *NeurIPS Datasets and Benchmarks*. https://arxiv.org/abs/2103.14749 ↩

9. Ratner, A., Bach, S. H., Ehrenberg, H., Fries, J., Wu, S. y Ré, C. (2017). Snorkel: Rapid Training Data Creation with Weak Supervision. *PVLDB*, 11(3), 269-282. https://doi.org/10.14778/3157794.3157797 ↩

.0. Little, R. J. A. y Rubin, D. B. (2019). *Statistical Analysis with Missing Data* (3.ª ed.). Wiley. ↩

.1. 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. ↩

.2. Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8), 707-710. ↩

.3. Broder, A. Z. (1997). On the Resemblance and Containment of Documents. *Proceedings. Compression and Complexity of Sequences 1997*, 21-29. https://doi.org/10.1109/SEQUEN.1997.666900 ↩

.4. Sainz, O., Campos, J. A., García-Ferrero, I., Etxaniz, J., de Lacalle, O. L. y Agirre, E. (2023). NLP Evaluation in Trouble: On the Need to Measure LLM Data Contamination for each Benchmark. *Findings of ACL: EMNLP 2023*. https://doi.org/10.18653/v1/2023.findings-emnlp.722 ↩

.5. Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley. ↩

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

.7. Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley. ↩

.8. scikit-learn. (2026). *train_test_split*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. Consultado el 6 de junio de 2026. ↩

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

20. Fleiss, J. L. (1971). Measuring Nominal Scale Agreement among Many Raters. *Psychological Bulletin*, 76(5), 378-382. https://doi.org/10.1037/h0031619 ↩

21. Krippendorff, K. (2004). Reliability in Content Analysis: Some Common Misconceptions and Recommendations. *Human Communication Research*, 30(3), 411-433. https://doi.org/10.1111/j.1468-2958.2004.tb00738.x ↩

22. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

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

24. Pandera. (2026). *Pandera Documentation*. https://pandera.readthedocs.io/en/stable/. Consultado el 6 de junio de 2026. ↩

25. AWS Labs. (2026). *Deequ: Unit Tests for Data*. https://github.com/awslabs/deequ. Consultado el 6 de junio de 2026. ↩

26. Soda. (2026). *What is Soda?* https://docs.soda.io/. Consultado el 21 de junio de 2026. ↩

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

28. Cleanlab. (2026). *Cleanlab Documentation*. https://docs.cleanlab.ai/. Consultado el 21 de junio de 2026. ↩

29. Evidently AI. (2026). *Evidently Documentation*. https://docs.evidentlyai.com/docs/library/overview. Consultado el 7 de junio de 2026. ↩

30. Monte Carlo. (2026). *Monte Carlo at a Glance*. https://docs.getmontecarlo.com/docs/architecture. Consultado el 21 de junio de 2026. ↩

31. Bigeye. (2026). *What is Bigeye?* https://docs.bigeye.com/docs/what-is-bigeye. Consultado el 21 de junio de 2026. ↩

32. Anomalo. (2026). *Product Overview*. https://www.anomalo.com/product-overview/. Consultado el 21 de junio de 2026. ↩

33. Datafold. (2026). *What's a Data Diff?* https://docs.datafold.com/data-diff/what-is-data-diff. Consultado el 21 de junio de 2026. ↩

Capítulo 03: Splits, muestreo y leakage: medir sin engañarse

Entrando en el tema

Hay una forma muy cómoda de engañarse con datos sin mala intención: partir el dataset, entrenar, mirar una métrica y respirar tranquilo. El problema es que la métrica puede estar respondiendo a otra pregunta. No “¿generaliza el sistema?”, sino “¿cuánto se parece test a lo que ya vio el sistema, el equipo o el pipeline?”.

En ciencia de datos para IA, el split es una decisión experimental. Decide qué mundo puede conocer el sistema, qué mundo usamos para ajustar decisiones y qué mundo queda reservado para comprobar si la conclusión se sostiene. Si esa frontera está mal puesta, todo lo posterior queda torcido: features, embeddings, RAG, evaluación de LLMs, slices, drift y gobernanza.

Este capítulo no va de memorizar `train_test_split`. Va de aprender a escribir una política de partición defendible: qué unidad separas, qué leaks buscas, qué transformaciones pueden aprender, cuándo el test deja de ser test y qué evidencia guardas para que otra persona pueda repetir la evaluación.

Qué deberías poder hacer al terminar

El capítulo anterior nos enseñó a bloquear un dataset con fallos de calidad. Pero un dataset puede pasar schema, licencias, etiquetas y duplicados básicos, y aun así medir mal. La razón suele estar en cómo lo partimos.

Un split no es un porcentaje. Es una promesa de evaluación. Cuando decimos `train`, `validation` y `test`, no estamos repartiendo filas al azar como quien reparte cartas. Estamos diciendo qué puede ver el sistema, qué usamos para ajustar decisiones y qué reservamos para comprobar si generaliza.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Explicar para qué sirve cada split.	No usas test para elegir prompt, modelo o umbral.
Elegir estrategia de muestreo según la pregunta.	No haces random split si hay tiempo, grupos o fuentes repetidas.
Detectar leakage por entidad, fuente, texto y tiempo.	Sabes buscar estudiantes, documentos o textos que cruzan splits.
Medir distribución de etiquetas por split.	No celebras un split si test pierde clases críticas.
Comparar estrategias de partición.	Puedes justificar random, estratificado, por grupo, temporal o híbrido.
Construir un manifiesto de split.	Dejas asignaciones y hallazgos versionados.

La gracia de esta lista está en que no puedes cumplirla solo con una función de librería. Puedes llamar a `train_test_split` correctamente y aun así medir mal si la unidad, el tiempo o la fuente documental no encajan con la pregunta. Un buen split tiene intención: dice qué mundo representa train, qué decisiones permite validation y qué realidad debe simular test.

La frase central del capítulo:

> Un split mide una pregunta. Si partes mal, la respuesta parece científica y no lo es.

La escena: una métrica que sube demasiado

Imagina que evaluamos un asistente académico. Hacemos un `train/test split` aleatorio y obtenemos una métrica preciosa. El sistema responde muy bien sobre matrículas, becas y pagos. Todo parece listo para enseñar.

Luego alguien mira los casos. Un estudiante aparece en train y test. Un documento fuente está repetido en ambos. Una consulta de test es casi igual a una de train. Además, algunos ejemplos de train tienen fecha posterior a ejemplos de test. La métrica no está midiendo generalización: está midiendo familiaridad.

Este es uno de los errores más caros porque no rompe el código. Al contrario: el código funciona, el reporte se ve profesional y la gráfica sube. Justo por eso hay que tratar los splits como ingeniería, no como una llamada rápida a una función.

Qué no es un split

Un split no es “60/20/20 y ya”. Esos números dicen cuántas filas hay, no si la partición responde bien a la pregunta. Si hay varios tickets de la misma persona, un random split puede poner unos en train y otros en test. Si hay documentos con versiones parecidas, puede partir la misma evidencia en dos. Si hay tiempo, puede entrenar con eventos posteriores y evaluar con eventos anteriores.

Tampoco es una garantía automática de independencia. Dos filas distintas pueden compartir entidad, fuente, plantilla, texto, anotador o evento. Para un modelo o un retriever, esa cercanía puede ser suficiente para inflar resultados.

Y test no es un lugar donde mirar muchas veces. Si usamos test para decidir prompts, elegir modelo, ajustar umbrales, cambiar chunking o seleccionar features, test deja de ser una medida final. Se convierte en otra validación, aunque sigamos llamándolo test.

Qué sí es un split de evaluación

Un split es una partición diseñada para responder a una pregunta de generalización. La notación siguiente no es una métrica propia del facsímil: es la forma básica de escribir una partición de conjuntos. El dataset completo se expresa como unión de subconjuntos, y esos subconjuntos no deben compartir filas:

$$D = D_{train} \cup D_{val} \cup D_{test}$$

con una condición de disjunción por pares:

$$D_i \cap D_j = \varnothing$$

para $i \neq j$, con $i, j \in \{train, val, test\}$. Dicho sin símbolos: una misma fila no debería estar a la vez en entrenamiento, validación y test.

SÍMBOLO	SIGNIFICADO	EJEMPLO
D	Dataset completo.	24 casos de soporte.
D_{train}	Datos que el sistema puede usar para aprender.	Casos anteriores usados para ajustar.
D_{val}	Datos para elegir decisiones durante desarrollo.	Elegir prompt, umbral o estrategia.
D_{test}	Datos reservados para medir al final.	Casos no usados para ajustar nada.
$D_i \cap D_j = \varnothing$	Los splits son disjuntos por pares.	Ninguna fila debería estar en dos splits.

La intersección vacía de filas es necesaria, pero no suficiente. También necesitamos evitar solapamiento de entidades, fuentes o información temporal cuando esas dimensiones afecten a la pregunta. En un dataset de IA, dos filas distintas pueden seguir estando contaminadas si pertenecen al mismo estudiante, salen del mismo documento o se generaron desde la misma traza.

Fecha de corte del estado del arte

Fecha de corte: 22 de junio de 2026. **Fuentes consultadas:** documentación oficial de scikit-learn sobre `train_test_split`, `GroupKFold`, `StratifiedGroupKFold`, `TimeSeriesSplit`, `Pipeline` y sus errores comunes; literatura sobre leakage en data mining y reproducibilidad de evaluación con machine learning; Wilson para intervalos de proporciones; Efron para bootstrap; Codd para pensar tablas y restricciones; DuckDB, dbt, Dagster y Airflow para aterrizar checks y orquestación; Databricks, Tecton y Feast para point-in-time joins; DVC y MLflow para trazabilidad de datasets; y el paper original de Retrieval-Augmented Generation para situar por qué, en RAG, documentos y consultas también forman parte de la evaluación.

`train_test_split` permite partir arrays o matrices en subconjuntos aleatorios de entrenamiento y test, con opciones como `test_size`, `train_size`, `random_state`, `shuffle` y `stratify`.[1] Esa función es útil, pero no decide por nosotros si un random split es correcto.

`GroupKFold` mantiene grupos no solapados entre folds, y `StratifiedGroupKFold` intenta preservar proporciones de clases sin partir grupos.[2][3] `TimeSeriesSplit` trabaja con particiones ordenadas en el tiempo, donde cada train es anterior al test correspondiente.[4]

Kaufman, Rosset, Perlich y Stitelman definieron leakage como la introducción de información sobre el objetivo que no debería estar disponible legítimamente para el modelo, y lo trataron como un error recurrente en proyectos reales y competiciones.[5] Kapoor y Narayanan revisaron leakage como una fuente importante de problemas de reproducibilidad en ciencia basada en machine learning.[6]

Lewis y colaboradores formalizaron RAG como una combinación de modelo paramétrico y memoria no paramétrica recuperada desde documentos.[7] Esa idea cambia la conversación sobre splits: no basta con partir preguntas, también hay que controlar documentos, versiones, chunks e índices de recuperación.

scikit-learn documenta `Pipeline` como una secuencia de transformadores y, en sus errores comunes, lo recomienda para evitar que estadísticas del test se filtren hacia el entrenamiento durante preprocesamiento, validación cruzada o búsqueda de hiperparámetros.[8][9] DVC se presenta como control de versiones para datos, modelos y experimentos, y MLflow documenta tracking de datasets y evaluation datasets para conservar linaje entre datos, evaluación y resultados.[10][11][12]

La lección estable es esta: las funciones de partición ayudan, pero la garantía viene de entender la estructura del dato.

Estrategias de muestreo

Antes de elegir una estrategia, debemos escribir la pregunta de evaluación. ¿Queremos saber si el sistema generaliza a nuevos casos de las mismas personas? ¿A nuevas personas? ¿A documentos futuros? ¿A otro periodo? ¿A clases raras? Cada pregunta pide un split distinto.

ESTRATEGIA	QUÉ PRESERVA	QUÉ PUEDE ROMPER
Random por fila	Proporciones aproximadas si hay suficientes datos.	Grupos, fuentes, tiempo y textos cercanos.
Estratificado por etiqueta	Distribución de clases.	Entidades repetidas o tiempo.
Por grupo	Entidades no solapadas.	Distribución de etiquetas o cronología.
Temporal	Simulación de futuro.	Grupos repetidos o clases raras.
Temporal por grupo	Tiempo y entidad a la vez.	Puede dejar alguna distribución en revisión.

No hay estrategia universal. Si todos los casos de una misma persona se parecen, necesitas grupo. Si el producto cambia con el tiempo, necesitas tiempo. Si hay clases muy raras, necesitas mirar cobertura por etiqueta. Si hay documentos comunes, quizá necesitas agrupar por fuente.

La estrategia correcta suele ser la que te obliga a renunciar a comodidad. Un random split da más filas repartidas y métricas más estables, pero puede estar midiendo familiaridad. Un split por grupo baja muestra efectiva, pero responde mejor si quieres generalizar a nuevas personas o empresas. Un split temporal puede ser más duro, pero se parece más a producción. En ingeniería de IA, una métrica más baja con un split honesto suele valer más que una métrica brillante con una partición ingenua.

Las fórmulas que sí conviene saber

Para medir si los splits conservan etiquetas, podemos comparar proporciones empíricas. Esto es la frecuencia relativa de una clase dentro de un split: no es una notación propia del facsímil, sino la forma estándar de estimar una distribución categórica a partir de conteos.

$$p_s(y = k) = \frac{n_{s,k}}{n_s}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
s	Split.	`train`, `validation`, `test`.
k	Etiqueta concreta.	`escalate`.
$n_{s,k}$	Ejemplos de la etiqueta k en el split s.	Casos `escalate` en test.
n_s	Total de ejemplos del split s.	5 casos de test.

Para comparar la distribución de un split con la global usamos distancia de variación total, una distancia habitual entre distribuciones de probabilidad.[13]

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

SÍMBOLO	SIGNIFICADO
P_i	Proporción global de la clase i.
Q_i	Proporción de la clase i en el split.
D_{TV}	Distancia entre ambas distribuciones.

Para textos cercanos usamos Jaccard, igual que en el capítulo anterior. La similitud de Jaccard viene de comparar intersección y unión de conjuntos.[14]

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

SÍMBOLO	SIGNIFICADO
A	Tokens del primer texto.
B	Tokens del segundo texto.
$\text{card}(A \cap B)$	Tokens compartidos.
$\text{card}(A \cup B)$	Tokens únicos totales.

Para leakage temporal no necesitamos inventar una métrica. Necesitamos una comprobación directa: si train contiene información posterior a la ventana que se quiere simular como futuro, la partición no sirve para esa pregunta.

COMPROBACIÓN	QUÉ MIRA	QUÉ IMPLICA SI FALLA
Fecha de train	Que las filas de aprendizaje son anteriores a test.	El modelo puede haber visto futuro.
Fecha de features	Que cada feature existía en el momento de decisión.	Hay leakage por variable calculada tarde.
Fecha de fuente	Que documentos y versiones pertenecen a la ventana correcta.	El RAG puede recuperar evidencia que no existía.
Fecha de etiqueta	Que la etiqueta no depende de un evento posterior no disponible.	La evaluación mide una ventaja imposible.

La lectura práctica de estas fórmulas y comprobaciones no es decorar el capítulo con símbolos. La distribución por split te dice si cada partición tiene casos suficientes para medir. La distancia de variación total te avisa cuando una partición deja de parecerse al conjunto que dice representar. Jaccard te ayuda a encontrar textos demasiado cercanos. Las fechas evitan que el sistema mire futuro. Ninguna decide sola, pero juntas obligan a mirar el split como un objeto medible y no como una partición hecha al vuelo.

Tipos de leakage que debe mirar un ingeniero

Leakage no siempre se ve como una fila duplicada. Muchas veces aparece como una relación escondida.

TIPO	CÓMO APARECE	QUÉ CHECK AYUDA
Por fila	Mismo `case_id` o mismo texto.	Duplicado exacto.
Por entidad	Misma persona, cliente, documento o dispositivo en train y test.	Group split.
Por fuente	Mismo `source_id` o versión documental.	Agrupar por fuente.
Por texto cercano	Preguntas casi iguales en splits distintos.	Jaccard, embeddings o revisión.
Temporal	Train ve eventos posteriores a test.	Split temporal.
De preprocesamiento	Escalado, imputación o selección de features hecha con todo el dataset.	Fit solo con train.
De decisión	Test se usa para elegir prompt, modelo o umbral.	Separar validation y test real.

El leakage de preprocesamiento merece una frase aparte. Si calculas la media de una columna con todo el dataset y luego escalas train y test, test ya influyó en train. La solución es sencilla conceptualmente: `fit` de transformaciones solo en train; `transform` en validation y test.

El split como contrato de evaluación

En ingeniería conviene dejar de hablar del split como una operación suelta y empezar a tratarlo como un contrato. Un contrato de split responde a preguntas que deberían quedar por escrito:

PREGUNTA	RESPUESTA QUE DEBE QUEDAR VERSIONADA
¿Qué decisión medimos?	Clasificador, RAG, prompt, modelo, retriever, umbral o pipeline completo.
¿Qué estrategia se eligió?	Random, estratificada, por grupo, temporal, temporal por grupo o una combinación propia.
¿Qué claves protegen la independencia?	`case_id`, `student_id`, `source_id`, `created_at`, `label`, `text`.
¿Qué puede hacerse con cada split?	Train entrena, validation decide, test mide, holdout confirma.
¿Qué está prohibido después de mirar test?	Elegir modelo, reescribir prompt, ajustar umbral, cambiar chunking o modificar el retriever.
¿Qué hashes demuestran reproducibilidad?	Hash del dataset, hash de la política y asignaciones por split.

El artefacto que recoge todo eso es el manifiesto de split. En el cuaderno del facsímil se genera como `output/split_manifest.json`. Un manifiesto mínimo debería contener:

```
{
  "policy_id": "support-split-policy-v1",
  "dataset": {
    "sha256": "..."
  },
  "split_contract": {
    "selected_strategy": "time_group_holdout",
    "gate": "review",
    "keys": {
      "group": "student_id",
      "source": "source_id",
      "time": "created_at"
    }
  }
}
```

El hash no es decoración. Si cambias una fila, una etiqueta o una fecha, el hash del dataset cambia. Si cambias una regla de la política, el hash de la política cambia. Eso permite reconstruir qué se midió, con qué reglas y por qué la métrica era defendible en ese momento.

En un equipo real, el manifiesto evita discusiones imposibles semanas después. Si alguien pregunta “¿por qué aquella métrica bajó al repetir el experimento?”, no deberíamos responder mirando una gráfica suelta. Deberíamos poder abrir el manifiesto, comprobar el hash del dataset, ver qué política estaba activa, saber qué estrategia asignó cada `case_id` y

revisar si test se usó solo para medir o también para decidir. Esa diferencia parece administrativa, pero es ingeniería: sin trazabilidad, una métrica deja de ser una medida y se convierte en una anécdota difícil de reproducir.

El manifiesto también ayuda a integrar evaluación en CI/CD. Puedes bloquear una ejecución si cambia el hash del dataset sin regenerar el split, si falta una predicción para un caso de test, si validation o test se usan para hacer `fit` de un transformador, o si una política nueva modifica los umbrales sin actualizar la decisión. En proyectos de IA, muchas regresiones no vienen de una línea de modelo, sino de una línea de datos que cambió sin que nadie la registrara.

Es un artefacto de ingeniería. Un manifiesto de split debe guardar, como mínimo, estas piezas:

PIEZA	QUÉ DEBE QUEDAR VERSIONADO	POR QUÉ IMPORTA
Dataset	Hash del snapshot evaluado.	Si cambia una fila, la métrica ya no describe lo mismo.
Política	Hash o versión de la regla de split.	Permite saber por qué se eligió una estrategia.
Estrategia	Random, estratificada, por grupo, temporal o combinada.	Explica qué generalización se pretendía medir.
Asignación	Qué IDs quedaron en train, validation y test.	Permite reproducir la evaluación.
Gate	`pass`, `review` o `block`, con motivos.	Convierte la partición en decisión auditable.

La métrica sin manifiesto es débil. Puede estar bien, pero no sabemos repetirla ni auditarla.

Point-in-time correctness: el split también vive en las features

Hasta aquí hemos hablado de filas, etiquetas, grupos y fuentes. Para un ingeniero de datos falta una pieza crítica: cada feature también tiene tiempo. No basta con que el `case_id` esté en el split correcto si las columnas usadas para entrenar o evaluar fueron calculadas con información posterior al momento en que ese caso existía.

En feature stores y pipelines de entrenamiento esto suele llamarse *point-in-time correctness*: construir cada fila de entrenamiento con los valores que estaban disponibles en el momento de decisión, no con el estado final de la tabla cuando alguien reejecuta el job.[151617]

La restricción formal es sencilla y poderosa. Si el caso e_i se decide en el instante t_i, una feature x_{ij} solo es válida si su fecha de disponibilidad no es posterior a t_i:

$$\text{available\_at}(x_{ij}) \leq t_i$$

SÍMBOLO	SIGNIFICADO	EJEMPLO DEL CUADERNO
e_i	Evento o caso que queremos entrenar/evaluar.	Caso `s023`.
t_i	Momento en que ese caso existía para el sistema.	`created_at = 2026-04-23`.
x_{ij}	Feature j usada para ese caso.	`open_cases_30d`.
$\text{available\_at}(x_{ij})$	Momento en que esa feature estaba disponible para usar.	`2026-04-20` o `2026-04-24`.

Esta desigualdad no es una métrica nueva del facsímil. Es una forma explícita de escribir la restricción de no mirar información que no estaba legítimamente disponible, justo el tipo de fuga que Kaufman y colaboradores describen como leakage. Si `s023` ocurrió el 23 de abril de 2026, una feature disponible el 20 de abril es válida; una feature recalculada o cargada el 24 de abril no puede usarse para simular la decisión del día 23.

Cuando hay varias versiones históricas de una feature, la operación correcta es un *as-of join*: para cada caso, elegir la versión más reciente que no mire al futuro. La condición temporal es:

$$\text{available\_at}(r) \leq t_i$$

Entre los registros que cumplen esa condición, elegimos el más reciente:

$$r_i^\star = \underset{r \in H_j(e_i),\ \text{available\_at}(r) \leq t_i}{\arg\max} \text{available\_at}(r)$$

Después usamos el valor de ese registro histórico como feature del caso:

$$x_j(e_i) = \text{value}(r_i^\star)$$

SÍMBOLO	SIGNIFICADO
$H_j(e_i)$	Historial de valores candidatos de la feature j para la entidad del caso e_i.
r	Una versión histórica concreta de esa feature.
$r_i^\star$	Registro histórico elegido para el caso e_i.
$\arg\max$	Selecciona la versión candidata con mayor fecha de disponibilidad permitida.

Ejemplo: si el caso `s020` es del 20 de abril y tenemos features del 19 y del 22, el as-of join debe quedarse con el valor del 19. El del 22 puede ser correcto como dato histórico, pero es incorrecto para ese evento. Si el job no distingue `event_time`, `feature_timestamp`, `available_at` e `ingested_at`, tarde o temprano terminará entrenando con futuro.

Esta es una diferencia importante entre ciencia de datos de notebook y ciencia de datos operable. En un notebook puedes leer la tabla final y calcular una métrica. En un sistema real necesitas saber qué snapshot, qué ventana y qué versión de feature existían cuando se tomó cada decisión.

Split materializado: que la partición sea una tabla

El split no debería vivir solo dentro de un script. Si una asignación decide qué filas entrenan y qué filas miden, debe poder auditarse como cualquier otra tabla de datos. En teoría relacional, Codd propuso tratar los datos como relaciones con atributos, claves y restricciones.[18] Para un split, esa idea baja a una regla muy concreta: cada caso debe tener una y solo una asignación activa en una versión de split.

Si A_v es la tabla de asignaciones de la versión v, la restricción de unicidad puede escribirse así:

$$\forall c \in D, \quad \operatorname{card}\{s : (c, s) \in A_v\} = 1$$

Aquí s pertenece al catálogo cerrado de splits: train, validation o test. La fórmula no pretende inventar una métrica nueva; expresa una restricción relacional muy básica: para cada caso hay exactamente una asignación en la versión del split.

SÍMBOLO	SIGNIFICADO	QUÉ COMPRUEBA
D	Dataset que queremos partir.	Todos los `case_id` del snapshot.
A_v	Tabla de asignaciones del split versión v.	`dataset_split_assignments.csv`.
c	Caso individual.	`s023`.
s	Split asignado.	`train`, `validation` o `test`.
$\text{card}(\cdot) = 1$	Hay exactamente una asignación.	No falta ni se duplica el caso.

El cuaderno materializa esa tabla como `output/dataset_split_assignments.csv`. No sustituye al manifiesto; lo complementa. El manifiesto cuenta la política, hashes y decisión. La tabla de asignaciones permite preguntar con SQL:

COLUMNA	POR QUÉ EXISTE
`case_id`	Clave de la fila evaluada.
`split`	Uso permitido de la fila.
`split_version`	Versión reproducible de la partición.
`policy_id`	Política que generó la asignación.
`dataset_sha256`	Snapshot de datos al que pertenece.
`policy_sha256`	Reglas exactas usadas para partir.
`assigned_at_utc`	Momento en que se generó la tabla.

En un equipo de datos, esta tabla puede vivir en DuckDB, BigQuery, Snowflake, Postgres o el almacén que uses. Lo importante no es la marca, sino que el split deje de ser memoria de un script y pase a ser un artefacto consultable, versionado y revisable por otra persona.

SQL de auditoría: pensar como data engineer

Una práctica útil es convertir los checks de split en consultas. dbt llama *data tests* a consultas que devuelven filas que incumplen una aserción; si no devuelven filas, la aserción pasa.[19] Dagster permite asociar checks a assets de datos.[20] Esa filosofía encaja muy bien aquí: un split sano no se “cree”, se comprueba.

En el cuaderno del facsímil hay dos niveles. Primero, `ops/run_sql_audit.py` ejecuta una auditoría con biblioteca estándar de Python para funcionar sin instalar nada. Segundo, `sql/split_audit_duckdb.sql` deja las consultas en estilo DuckDB para que un alumno pueda llevarlas a una base analítica real. DuckDB permite leer CSV directamente desde SQL, lo que lo convierte en una herramienta cómoda para prototipar auditorías locales.[21]

Las preguntas que hacemos con SQL son muy de ingeniería:

PREGUNTA	QUÉ REVELA
¿Hay `case_id` duplicados en la tabla de split?	La misma fila podría entrenar y evaluar.
¿Falta algún `case_id` del dataset?	La evaluación no cubre todo el snapshot declarado.
¿Un `student_id` aparece en train y test?	Leakage por entidad.
¿Un `source_id` aparece en train y test?	Leakage por fuente documental.
¿Hay features posteriores al `created_at` del caso?	Un join ingenuo puede mirar futuro.
¿El test cubre las etiquetas obligatorias?	La métrica global puede esconder clases ausentes.

La parte importante es el matiz de las features. Que existan features posteriores a un caso no es malo por sí mismo: los datos históricos pueden seguir creciendo. Lo malo es que el join elija esas filas. Por eso el reporte distingue entre "hay candidatas futuras que un join ingenuo podría coger" y "el as-of join seguro selecciona la última feature disponible antes de la decisión".

Este es un buen ejemplo de práctica que sí se lleva al trabajo. Un SQL de auditoría puede vivir en dbt, en DuckDB local, en BigQuery o en Snowflake, pero la pregunta es la misma: "¿qué filas violan mi contrato?". Si la consulta devuelve casos, no tienes una opinión; tienes evidencia revisable. Esa forma de trabajar acerca la evaluación de IA a ingeniería de datos profesional.

Backfills y re-splits

Un *backfill* es un reproceso histórico. Puede aparecer porque llegó tarde una fuente, se corrigieron etiquetas, cambió un parser, se rehidrató una tabla o se recalculó una feature. En ciencia de datos para IA, un backfill no es una tarea inocente: puede cambiar el dataset sobre el que se entrenó, el split que se había congelado o la métrica que se comunicó.

El error común es "vuelvo a ejecutar y ya está". Eso borra la pregunta original. Si el snapshot cambia, la versión del split también debe revisarse. Si una etiqueta cambia, una métrica anterior puede quedar invalidada. Si una feature se recalcula con una fecha de disponibilidad

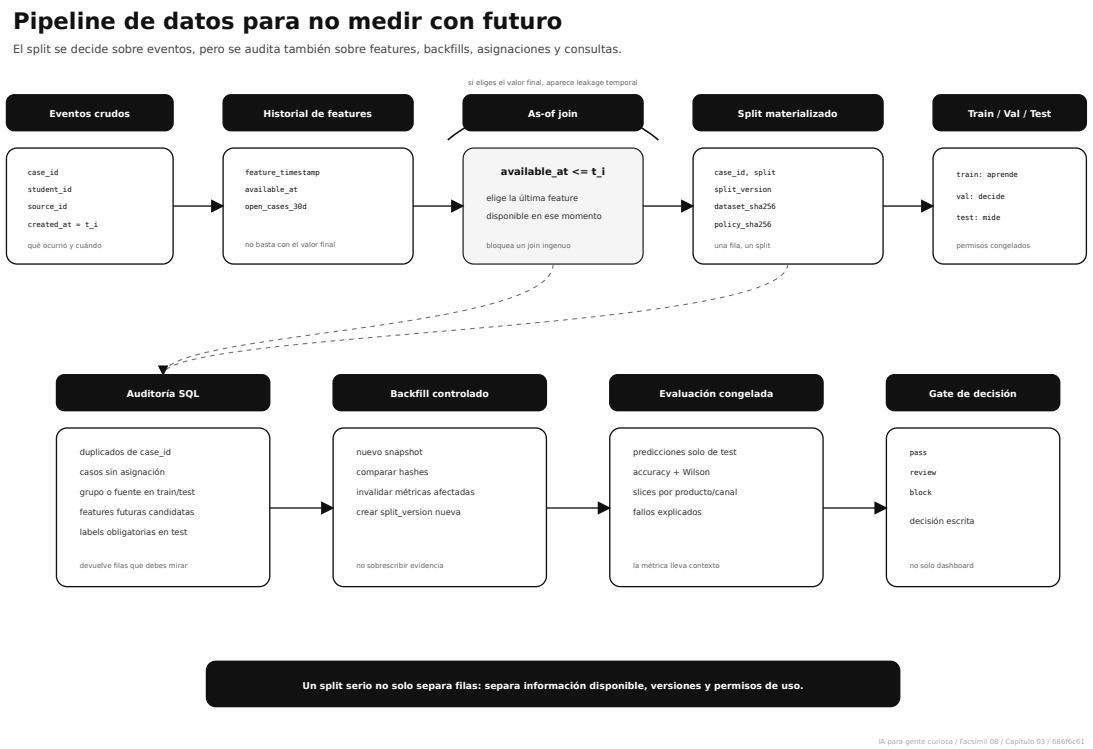
distinta, puede introducir leakage temporal aunque el código siga pasando.

Una política práctica:

CAMBIO	QUÉ HACER ANTES DE MEDIR
Llegan filas históricas tarde.	Crear nuevo snapshot y comparar asignaciones.
Cambia una etiqueta.	Marcar métricas afectadas y regenerar reporte.
Cambia una fuente documental.	Revisar grupos por `source_id` o versión.
Cambia una feature.	Auditar `available_at` y repetir as-of join.
Cambia la política de split.	Crear `split_version` nueva, no sobrescribir la anterior.
Cambia el parser o la limpieza.	Recalcular hashes y revisar duplicados/leakage textual.

Airflow documenta scheduling sensible a datasets: tareas que se disparan cuando cambian datasets upstream.[22] Eso es útil, pero no suficiente. Que un dataset cambie y dispare un pipeline no significa que la evaluación siga siendo comparable. Necesitas una decisión explícita: mantener split versionado, crear re-split o bloquear hasta revisión.

Pipeline técnico: de eventos a evaluación reproducible



Pipeline de ingeniería de datos para auditar splits: eventos, features temporales, as-of join, asignaciones materializadas, SQL, backfills y gate final.

Leakage de preprocesamiento

Una fuga clásica no aparece en el split, sino en el pipeline. Suele pasar cuando hacemos esto:

1. Cargar todo el dataset.
2. Normalizar, imputar, seleccionar variables o crear vocabulario.
3. Partir en train, validation y test.
4. Entrenar y medir.

El problema está en el paso 2. Si una transformación aprende algo del dataset completo, validation y test ya han participado en el entrenamiento, aunque el modelo principal no los haya visto.

La versión correcta es un patrón de ejecución, no una ecuación:

PASO	QUÉ SE PERMITE APRENDER	DÓNDE SE APLICA DESPUÉS
`fit` de la transformación	Parámetros de escalado, imputación, vocabulario, PCA o calibración.	Solo con `train`.
Congelar parámetros	La media, vocabulario, componentes o reglas aprendidas.	Se versionan con el pipeline.
`transform`	No aprende parámetros nuevos.	Se aplica a `train`, `validation` y `test` con lo aprendido en `train`.

La regla práctica:

TRANSFORMACIÓN	QUÉ APRENDE	DÓNDE SE HACE `FIT`	QUÉ SE APLICA A VALIDATION/TEST
Normalización	Media, desviación, mínimos o máximos.	Solo train.	`transform` con parámetros de train.
Imputación	Media, moda, percentil o modelo auxiliar.	Solo train.	Relleno con reglas aprendidas en train.
Selección de features	Variables que parecen predictivas.	Solo train, o dentro de cada fold.	Mismas features elegidas.
PCA	Componentes y proyección.	Solo train.	Misma proyección.
TF-IDF	Vocabulario e IDF.	Solo train.	Misma matriz de vocabulario.
Oversampling	Ejemplos sintéticos o pesos.	Solo train.	Nunca se sintetiza test.

La frase que debería quedar pegada a la pantalla: primero partes, luego aprendes transformaciones.

En el cuaderno del facsímil, esta idea se convierte en un script ejecutable: `ops/preprocessing_fit_audit.py`. El script no entrena un modelo grande; hace algo más pequeño y muy útil para aprender: compara el vocabulario que obtendrías si ajustas un vectorizador con solo train frente al vocabulario que obtendrías usando todo el dataset. Si el vocabulario global aprende palabras que solo aparecen en validation o test, acabas de demostrar una fuga de preprocesamiento con un ejemplo que cabe en una terminal.

Por ejemplo, con el split recomendado del cuaderno, el vectorizador ajustado solo con train no conoce algunos términos que aparecen en test, como `convocatoria`, `abonado`, `confirmacion` o `bloqueada`. Eso es normal: test representa futuro o datos reservados. Si ajustas el vectorizador con todo el dataset antes de partir, esas palabras entran en el vocabulario desde

el principio. El modelo todavía no ha visto la etiqueta de test, pero su representación ya fue preparada con información del test. En texto, ese detalle es importante porque vocabulario, IDF, normalizadores y reglas de tokenización forman parte de lo que el sistema aprende.

Puedes ejecutarlo así:

```
# Desde la carpeta extraída del ZIP:
python3 ops/split_audit.py --write
python3 ops/preprocessing_fit_audit.py --write
cat output/preprocessing_fit_decision.md
```

La salida no pretende decir “este vocabulario es malo”. Dice algo más concreto: si quieres medir honestamente, el `fit` del vectorizador debe ocurrir después del split y solo con train. Luego aplicas ese mismo vectorizador a validation y test. Ese patrón se traslada igual a `StandardScaler`, imputadores, PCA, selección de features, calibración, oversampling o cualquier transformación que aprenda parámetros.

RAG y LLM eval también tienen splits

En un clasificador tabular, solemos pensar en filas. En RAG y evaluación de LLMs hay más piezas:

OBJETO	POR QUÉ PUEDE CONTAMINAR LA EVALUACIÓN
Documento	El mismo documento puede generar chunks, preguntas y respuestas en varios splits.
Chunk	Dos chunks hermanos pueden contener casi la misma evidencia.
Pregunta sintética	Si se genera desde un documento, hereda su grupo.
Respuesta esperada	Si acaba en trazas de desarrollo, el test deja de ser final.
Prompt few-shot	Ejemplos demasiado parecidos al test facilitan la tarea artificialmente.
Índice de recuperación	Si mezcla corpus de desarrollo y test, la medición no separa conocimiento.
Herramientas y trazas	Logs de ejecución pueden contener soluciones, IDs o rutas documentales.

Para RAG, la regla no debería ser “parto preguntas”. Debería ser un contrato documental:

SI COMPARTEN...	DEBEN QUEDARSE JUNTOS CUANDO MIDES...	EJEMPLO
`document_id`	Generalización a documentos nuevos.	Chunks hermanos del mismo PDF.
`source_id` o URL canónica	Recuperación sobre fuentes no vistas.	Varias preguntas generadas desde la misma normativa.
Versión temporal	Robustez ante cambios de documento.	Reglamento 2025 frente a reglamento 2026.

Si dos chunks vienen del mismo documento, no deberían terminar uno en train y otro en test salvo que la evaluación esté diseñada explícitamente para medir recuperación sobre documentos compartidos. Y si ese caso existe, debe declararse, porque ya no estamos midiendo generalización documental.

En evaluación de LLMs hay otra trampa: mirar fallos de test para mejorar el prompt. La primera vez puede ser análisis. La segunda empieza a parecer ajuste. Si cambiamos el prompt, el modelo, el formato de salida, el sistema de herramientas o la rúbrica porque vimos test, test ya no es final. Necesitamos validation o un nuevo holdout.

Un ejemplo operativo: supón que tienes 200 preguntas para evaluar un RAG de normativa universitaria. Si generas 100 preguntas desde un documento de becas y luego repartes esas preguntas al azar, algunas acabarán en train, otras en validation y otras en test. El modelo o el equipo no ven exactamente la misma pregunta, pero sí ven el mismo documento, los mismos términos y a veces la misma respuesta escrita de otra manera. Para medir generalización documental, eso no vale. El grupo no debería ser la pregunta: debería ser el documento, la versión del documento o la fuente canónica.

Otro caso frecuente: usas ejemplos few-shot en el prompt. Esos ejemplos también tienen que salir de train o validation, no del test. Si al mirar el test descubres que el modelo falla en “matrícula bloqueada” y copias un ejemplo parecido al prompt, has convertido el test en una herramienta de ajuste. La práctica correcta no es dejar de aprender del error; es mover esa decisión a validation y reservar un nuevo holdout si quieres comunicar una cifra final.

Validación cruzada avanzada

La validación cruzada no arregla por sí sola un split mal pensado. Solo repite una regla de partición varias veces. Si la regla parte grupos o mezcla futuro, el problema se repite en cada fold.

MÉTODO	CUÁNDO SIRVE	RIESGO SI SE USA SIN PENSAR
`KFold`	Datos independientes y suficientes.	Parte entidades repetidas.
`StratifiedKFold`	Clasificación con clases desbalanceadas.	Conserva etiquetas, pero puede mezclar grupos.
`GroupKFold`	Hay usuarios, estudiantes, clientes, pacientes, documentos o dispositivos.	Puede dejar folds con etiquetas descompensadas.
`StratifiedGroupKFold`	Necesitas grupos no compartidos y proporciones de clases razonables.	No siempre puede satisfacer ambas cosas si el dataset es pequeño.
`TimeSeriesSplit`	Evaluación hacia futuro.	No protege entidades repetidas por sí solo.
Validación cruzada anidada	Separar selección de hiperparámetros y estimación final.	Es más cara y exige disciplina con pipelines.

La validación cruzada anidada se usa cuando queremos seleccionar hiperparámetros sin engañar la estimación. La idea operativa se entiende mejor como dos bucles con papeles distintos:

BUCLE	PAPEL	QUÉ NO DEBE HACER
Externo	Estimar rendimiento final en folds que no han decidido la configuración.	Elegir hiperparámetros mirando su propio test.
Interno	Elegir hiperparámetros dentro del train de cada fold externo.	Usar datos del fold externo reservado.

Si usas el mismo fold para elegir y medir, el resultado tiende a ser optimista. No porque nadie haya hecho nada extraño, sino porque el proceso de selección ya miró esa señal.

Negativos difíciles, ranking y recuperación

En RAG, ranking y clasificación semántica, los ejemplos negativos importan mucho. Un negativo fácil es un caso claramente distinto. Un negativo difícil es uno que se parece mucho al positivo pero requiere distinguir un detalle.

CASO	POSITIVO	NEGATIVO DIFÍCIL
Beca	“requisitos de beca general”	“requisitos de beca de movilidad”
Pago	“justificante de pago pendiente”	“pago duplicado reclamado”
Matrícula	“plazo ordinario de matrícula”	“ampliación de matrícula fuera de plazo”

Los negativos difíciles enseñan y evalúan mejor, pero también pueden contaminar si se generan mezclando splits. La regla operativa:

1. Si generas negativos antes de partir, agrupa por entidad y fuente para que no crucen splits.
2. Si generas negativos después de partir, hazlo dentro de cada split.
3. Si usas embeddings para buscar negativos, no uses el índice de test para seleccionar ejemplos de train.
4. Si ajustas un reranker, mide con consultas y documentos que no hayan participado en su selección.

Para un retriever, una evaluación mínima debería reportar:

MÉTRICA	QUÉ PREGUNTA RESPONDE
`recall@k`	¿El documento correcto aparece entre los k primeros?
`MRR`	¿A qué altura aparece el primer resultado útil?
Precisión de contexto	¿Cuánto contexto recuperado es realmente relevante?
Cobertura por fuente	¿El sistema recupera bien todos los tipos de documento?
Latencia	¿La recuperación cabe en el presupuesto operativo?

Este punto prepara el capítulo de embeddings y los capítulos de RAG: la evaluación de recuperación también empieza por un split honesto.

Tamaño mínimo, intervalos y slices

Un test pequeño puede ser honesto y aun así poco informativo. Si medimos accuracy con 20 ejemplos, pasar de 15 a 16 aciertos cambia la métrica de 75% a 80%. Eso no es una mejora estable; puede ser ruido muestral.

Para una proporción binomial, como accuracy cuando cada caso cuenta como acierto o fallo, el intervalo de Wilson suele ser más prudente que el intervalo normal simple cuando el test es pequeño.[23]

$$IC_{Wilson} = \frac{\hat{p} + \frac{z^2}{2n} \pm z \sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}$$

SÍMBOLO	SIGNIFICADO
$\hat{p}$	Métrica como proporción: accuracy, tasa de acierto, recall binario.
n	Número de ejemplos del test.
z	Cuantil normal asociado al nivel de confianza; para 95% se usa aproximadamente 1.96.
IC_{Wilson}	Intervalo de Wilson para la proporción.

Ejemplo del cuaderno: si $\hat{p} = 0.75$ y $n = 4$, el intervalo de Wilson al 95% queda aproximadamente entre 0.30 y 0.95. Ese rango es enorme. La lectura no es “el sistema está entre 30% y 95% y ya está”, sino “con cuatro casos no hay base suficiente para cerrar una afirmación fuerte”.

Para métricas más complejas, bootstrap suele ser una opción práctica: remuestrear el test muchas veces y observar la distribución de la métrica.[24] Pero incluso bootstrap necesita una pregunta honesta: si el test está contaminado, remuestrear no arregla la contaminación.

Además de mirar la media global, hay que mirar slices. Un sistema puede tener buen resultado global y fallar justo en el segmento que importa.

SLICE	POR QUÉ MIRARLO
`product`	Matrícula, becas, pagos, horarios o prácticas pueden tener dificultad distinta.
`channel`	Portal, email y chat no generan el mismo lenguaje.
`label`	Las clases raras suelen desaparecer en tests pequeños.
`source_id`	Algunas fuentes documentales son más ambiguas.
Fecha	Cambios de periodo pueden alterar vocabulario y reglas.
Criticidad	No todos los errores cuestan lo mismo.
Idioma o región	Si existen, pueden cambiar distribución y estilo.

La pregunta técnica no es solo “¿cuánto acierta?”. También es: “¿dónde falla, con cuánta evidencia y qué decisión puedo tomar sin engañarme?”.

El segundo script práctico del cuaderno trabaja justo esta parte: `ops/evaluate_test_slices.py`. Lee `data/model_predictions.csv`, toma del manifiesto los IDs asignados a test y evalúa solo esos casos. Esto es importante: no deja que se cuele una predicción de train o validation en la métrica final. Después calcula accuracy, intervalo de Wilson, latencia y slices por `product`, `channel` y `label`.

Con el dataset pequeño del capítulo, el resultado esperado es deliberadamente incómodo: accuracy de 0.75 con solo 4 casos de test, un intervalo de Wilson muy ancho, un fallo en el caso `s023` y una decisión `review_test_too_small`. Eso enseña una lección que muchos proyectos aprenden tarde: una métrica global puede sonar bien y seguir sin ser suficiente para tomar una decisión fuerte. Si el único caso `escalate` del test falla, la media global no cuenta toda la historia. Si cada producto tiene un único ejemplo, ningún slice tiene evidencia suficiente. El reporte no bloquea por capricho; te obliga a mirar la métrica con tamaño, contexto y coste del error.

Puedes ejecutarlo así:

```
# Desde la carpeta extraída del ZIP:
python3 ops/evaluate_test_slices.py --write
cat output/evaluation_slice_decision.md
python3 -m json.tool output/evaluation_slice_report.json
```

Este ejemplo sí se puede reutilizar. Cambias `data/model_predictions.csv` por las predicciones de tu modelo, mantienes `case_id`, `predicted_label`, `confidence` y `latency_ms`, y el script usa el manifiesto para decidir qué pertenece a test. Si tu proyecto mide otra cosa, cambias la función de métrica, pero mantienes el contrato: IDs versionados, split congelado, slices explícitos y decisión documentada.

Herramientas y patrones que sí ayudan

En este capítulo no basta con decir “usa una librería”. Las herramientas ayudan cuando codifican la política correcta. Si la política está mal, la herramienta solo ejecuta más rápido el error.

NECESIDAD	HERRAMIENTA O PATRÓN	QUÉ APORTA	QUÉ DEBES VIGILAR
Partir datos tabulares o arrays	`train_test_split`, `GroupKFold`, `StratifiedGroupKFold`, `TimeSeriesSplit`	Implementa estrategias conocidas y reproducibles.	La función no sabe cuál es tu unidad experimental real.
Evitar leakage de preprocesamiento	`Pipeline` y transformadores con `fit` / `transform`	Encadena pasos para que cada fold entrene transformadores solo con train.	Un `fit` manual fuera del pipeline puede volver a contaminarlo todo.
Versionar datos y manifiestos	DVC, lakeFS, hashes propios o registro interno	Permite reconstruir qué snapshot produjo qué métrica.	Versionar archivos no sustituye escribir una política de split.
Auditar como tabla	DuckDB, SQL en almacén analítico, dbt data tests	Convierte reglas de split en consultas que devuelven filas problemáticas.	Un test SQL mal planteado puede pasar aunque la pregunta de evaluación sea equivocada.
Orquestar cambios	Dagster asset checks, Airflow data-aware scheduling	Conecta cambios de datasets con checks y pipelines.	Reejecutar no basta: un backfill puede exigir re-split o invalidar métricas.
Construir features temporales	Feast, Tecton, Databricks Feature Store	Ayuda a hacer as-of joins y point-in-time correctness.	Si `available_at` está mal definido, la herramienta no puede adivinar la realidad.
Registrar evaluaciones	MLflow Dataset Tracking o evaluation datasets	Conecta dataset, run, métrica y artefactos de evaluación.	Si el test se usó para decidir, el tracking debe decirlo.
Integrar en CI/CD	`make test`, GitHub Actions, GitLab CI, Buildkite	Bloquea cambios cuando faltan outputs, predicciones o manifiesto.	Un CI que solo mira que el script termina no evalúa calidad semántica.

El patrón profesional es este: primero escribes la pregunta de evaluación y la política de split; después eliges herramienta. Si empiezas por la herramienta, acabarás aceptando sus valores por defecto como si fueran criterio científico.

Política de test y holdout final

La política sencilla:

SPLIT	PERMISO
Train	Aprender.
Validation	Elegir.
Test	Medir una decisión cerrada.
Holdout final	Confirmar una conclusión importante.

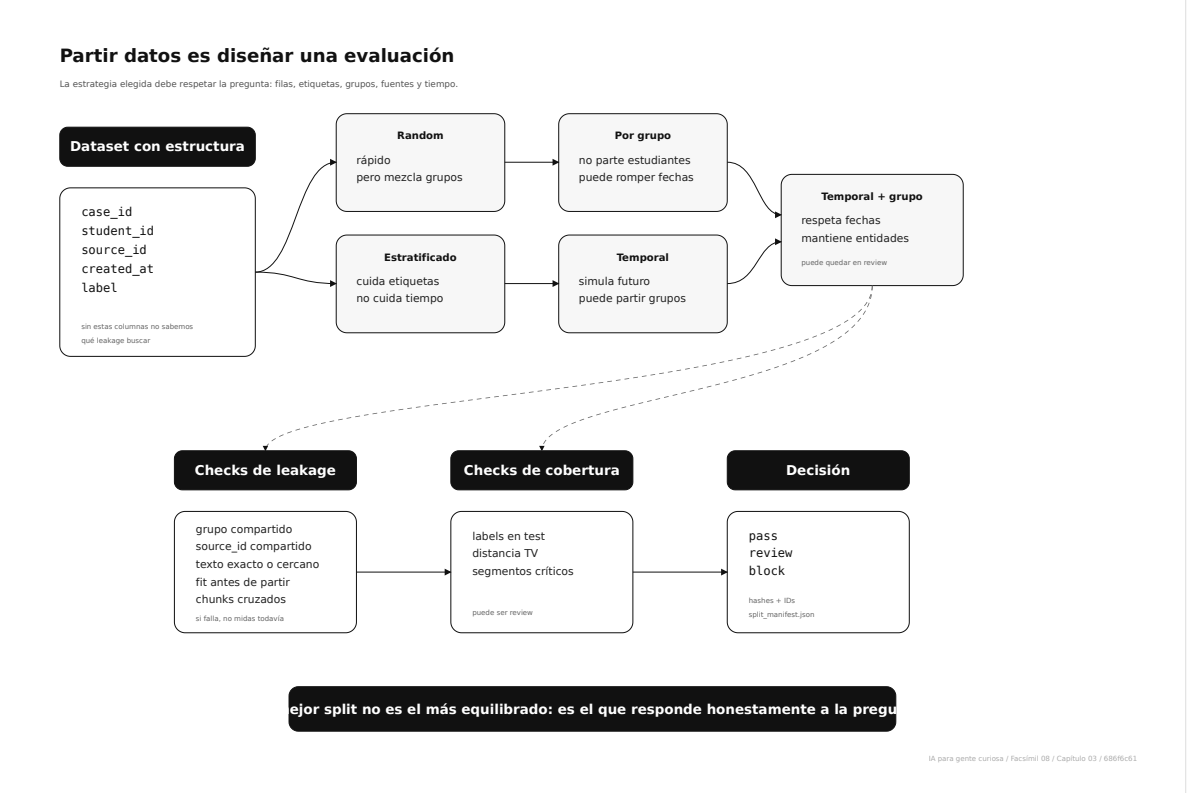
Cuando test se usa para decidir, deja de ser test final. En un proyecto real eso no debería vivirse como culpa, sino como trazabilidad: se anota y se crea una nueva partición final si hace falta.

Un ejemplo concreto:

1. Entrenas tres modelos con train.
2. Eliges el mejor con validation.
3. Mides una vez con test.
4. Ves que falla en “becas”.
5. Cambias el prompt, el chunking o el retriever.
6. La siguiente medición ya no debería venderse como test puro.

La solución profesional es reservar un holdout final para decisiones importantes o crear una nueva versión de evaluación. No siempre hace falta en un ejercicio pequeño; sí hace falta cuando vas a comunicar rendimiento, comparar sistemas o tomar una decisión con impacto.

Anatomía de una partición que no se engaña



Una partición seria combina estrategia, checks de leakage, cobertura y una decisión documentada.

En el día a día

Un equipo profesional debería discutir el split antes de mirar resultados. Si elegimos estrategia después de ver métricas, ya estamos contaminando la decisión. La pregunta correcta es: “¿qué uso real queremos simular?”.

SITUACIÓN	SPLIT RAZONABLE	POR QUÉ
Tickets de los mismos clientes	Por cliente o cuenta.	Evita que el sistema vea estilos repetidos.
Documentación viva	Temporal por versión.	Evalúa contra futuro documental.
Datasets médicos o académicos por persona	Por sujeto o estudiante.	Evita que la misma entidad cruce particiones.
Clases raras	Estratificado con revisión manual.	Evita perder casos críticos en test.
Logs de producto	Temporal.	Simula despliegue hacia adelante.

La regla profesional: si una entidad puede generar varias filas parecidas, no partas por fila hasta demostrar que no contamina.

Por qué debería importarte

Una mala partición puede hacer que un modelo mediocre parezca excelente. También puede hacer que un modelo bueno parezca inestable si el test no cubre clases importantes. En ambos casos, la decisión técnica se apoya en una medición defectuosa.

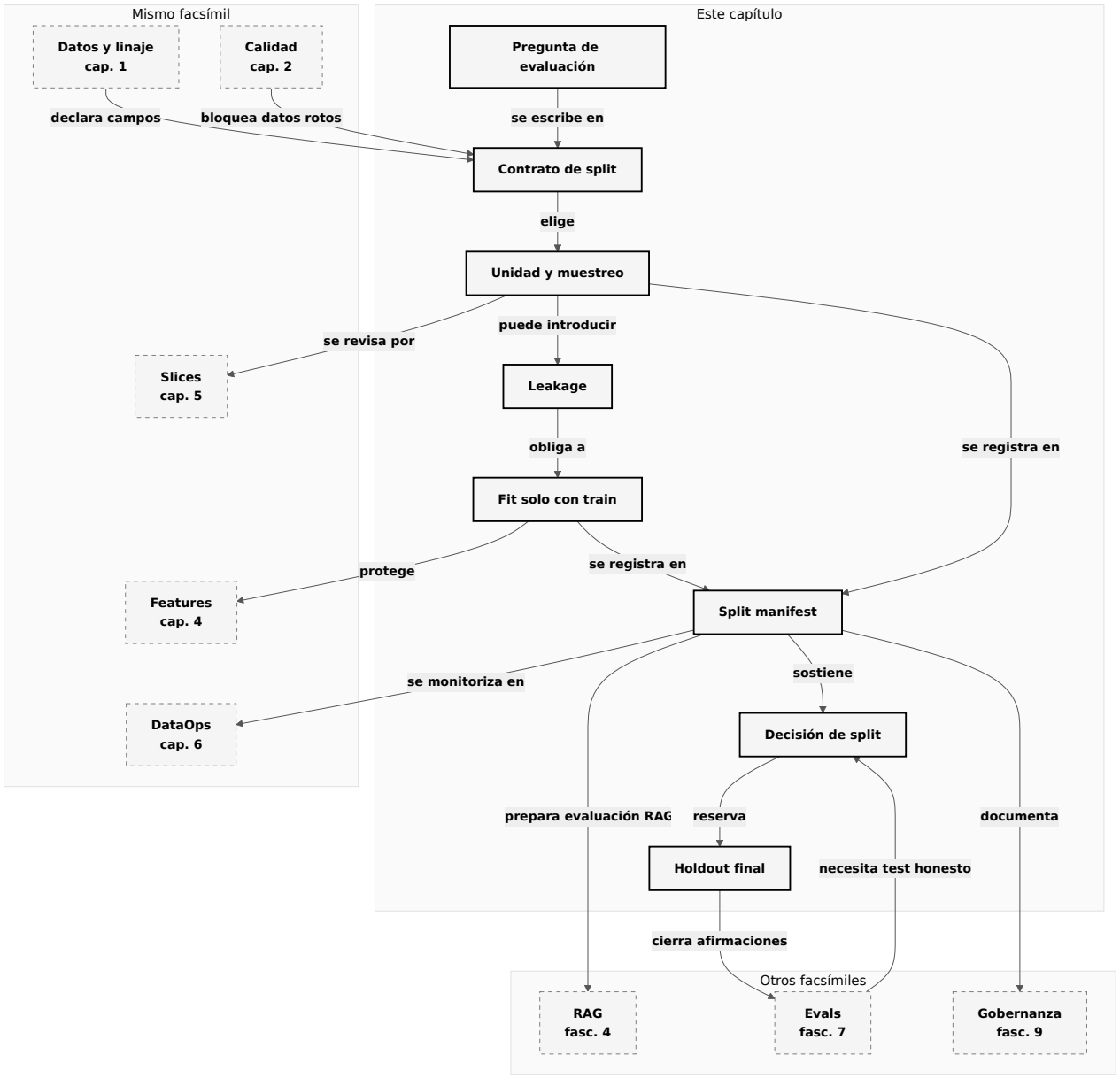
Este capítulo importa porque casi todo lo que sigue depende de aquí: features, embeddings, slices, drift, análisis causal y gobernanza. Si test no es honesto, los capítulos posteriores están midiendo sobre una base torcida.

Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Confundir porcentaje con validez	60/20/20 parece profesional.	Preguntar qué decisión simula cada split.
Estratificar y olvidarme de grupos	Las clases quedan bonitas.	Revisar `student_id`, cliente, fuente o documento.
Hacer split después de transformar	El pipeline parece más cómodo.	Separar primero; ajustar transformaciones solo en train.
Mirar test demasiadas veces	Es tentador ajustar contra la métrica final.	Usar validation para decisiones y test para cierre.
Ignorar el tiempo	Random parece estable.	Usar split temporal si producción será futura.
Partir preguntas RAG sin agrupar documentos	Parece que cada pregunta es independiente.	Agrupar por documento, versión, fuente o URL canónica.
Hacer negativos difíciles con todo el índice	Sale una evaluación más dura, pero mezclada.	Generarlos dentro de cada split o con grupos congelados.
No guardar manifiesto	El reporte parece suficiente.	Versionar hashes, política, estrategia e IDs por split.
Celebrar una métrica con test pequeño	El número tiene demasiada varianza.	Acompañar con tamaño, intervalo y slices.

Cómo encaja todo

Este capítulo conecta calidad de datos con evaluación. El capítulo 02 detectaba fallos dentro del dataset; este capítulo pregunta si la partición permite medir sin engañarse. El capítulo 04 usará estos splits para hablar de features y embeddings: ahí veremos que una transformación también puede introducir leakage si se ajusta con todo el dataset.


Mismo facsímil
Datos y linaje
cap. 1
Calidad
cap. 2
declara campos
bloquea datos rotos
Slices
cap. 5
Features
cap. 4
DataOps
cap. 6
Este capítulo
Pregunta de evaluación
se escribe en
Contrato de split
elige
Unidad y muestreo
se revisa por
puede introducir
Leakage
obliga a
Fit solo con train
protege
se registra en
se registra en
Split manifest
se monitoriza en
sostiene
Decisión de split
prepara evaluación RAG
reserva
documenta
Holdout final
necesita test honesto
cierra afirmaciones
Otros facsímiles
RAG
fasc. 4
Evals
fasc. 7
Gobernanza
fasc. 9

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Split	Partición de un dataset con finalidad concreta.
Train	Split que puede usarse para aprender o ajustar.
Validation	Split para elegir decisiones durante desarrollo.
Test	Split reservado para medir al final.
Holdout	Conjunto separado que no se usa durante ajuste.
Estratificación	Mantener proporciones de etiquetas o segmentos.
Split por grupo	Mantener todas las filas de una entidad juntas.
Split temporal	Respetar orden cronológico.
Leakage	Información indebida que contamina la evaluación.
Purga	Separar o retirar ejemplos demasiado cercanos.
Distancia TV	Medida de diferencia entre distribuciones.
Manifiesto de split	Documento técnico con hashes, estrategia, claves, permisos y asignaciones.
Leakage de preprocesamiento	Fuga causada por transformaciones ajustadas antes de partir o usando splits reservados.
RAG leakage	Fuga en documentos, chunks, preguntas, respuestas esperadas, prompts o índices.
Negativo difícil	Ejemplo muy parecido al positivo, pero con etiqueta o respuesta distinta.
Holdout final	Partición reservada para confirmar una conclusión después de cerrar decisiones.
Slice	Subconjunto relevante del test: producto, canal, etiqueta, fuente, fecha o criticidad.
Intervalo de Wilson	Intervalo de confianza para una proporción binomial, útil cuando el test es pequeño.
Pipeline de preprocesamiento	Cadena de transformaciones donde cada paso que aprende parámetros debe hacer `fit` solo con train.
Dataset tracking	Registro de datasets, hashes, splits, runs y resultados para poder reconstruir una evaluación.

TÉRMINO	DEFINICIÓN BREVE
Point-in-time correctness	Garantía de que cada feature usada estaba disponible en el momento de decisión del caso.
As-of join	Unión temporal que elige el último valor válido antes o en la fecha del evento.
Split materializado	Tabla versionada que asigna cada caso a un único split.
Backfill	Reproceso histórico que puede cambiar snapshots, features, etiquetas o métricas.
Re-split	Nueva versión de partición creada cuando cambia el dato, la política o la pregunta de evaluación.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Por qué un split no es solo un porcentaje?
2. ¿Qué diferencia hay entre train, validation y test?
3. ¿Cuándo usarías split por grupo?
4. ¿Cuándo usarías split temporal?
5. ¿Por qué una estrategia estratificada puede seguir contaminada?
6. ¿Qué mide Jaccard al detectar duplicados entre splits?
7. ¿Qué significa `future_train_vs_test`?
8. ¿Por qué `time_group_holdout` queda en `review` y no en `pass`?
9. ¿Qué artefacto explica los hallazgos de leakage?
10. ¿Cómo evitarías leakage de preprocesamiento?
11. ¿Por qué un documento RAG puede contaminar varias preguntas?
12. ¿Qué pasa si miras test para cambiar el prompt?
13. ¿Qué diferencia hay entre validation, test y holdout final?
14. ¿Qué debería guardar un `split_manifest.json`?
15. ¿Cuándo usarías `StratifiedGroupKFold` en vez de `KFold`?
16. ¿Por qué una métrica global puede esconder fallos por slice?
17. ¿Por qué usamos Wilson cuando el test es pequeño?
18. ¿Qué demuestra `preprocessing_fit_audit.py`?
19. ¿Por qué `evaluate_test_slices.py` filtra por IDs del manifiesto?

20. ¿Qué significa que una feature cumpla `available_at <= created_at`?
21. ¿Por qué un as-of join protege contra leakage temporal?
22. ¿Qué comprueba `dataset_split_assignments.csv` que el manifiesto no muestra tan cómodamente?
23. ¿Qué harías si un backfill cambia etiquetas o features históricas?
24. ¿Qué mirarías en `sql_split_audit_decision.md` antes de confiar en el split?
25. ¿Qué cambiarías en `model_predictions.csv` para usarlo con tu propio modelo?

En resumen

IDEA	QUÉ TE LLEVAS
Un split mide una pregunta.	La estrategia depende del uso real.
Random no siempre es inocente.	Puede mezclar grupos, fuentes y tiempo.
Estratificar no basta.	Conserva labels, pero no protege entidades.
El tiempo cambia la evaluación.	Entrenar con futuro infla resultados.
La mejor estrategia puede quedar en review.	Evitar leakage no garantiza cobertura perfecta.
El split debe versionarse.	Asignaciones, hallazgos y decisión son artefactos.
Preprocesar también aprende.	Todo `fit` que aprende parámetros debe usar train.
RAG añade nuevas fronteras.	Documentos, chunks, prompts y respuestas esperadas también se separan.
Test no decide.	Si test decide, necesitas validation nueva o holdout final.
Los negativos difíciles se controlan.	Son útiles, pero deben respetar grupos y particiones.
El tamaño importa.	Un test pequeño necesita intervalo, cautela y lectura por slices.
Las herramientas no deciden por ti.	`Pipeline`, DVC o MLflow ayudan si la política de split ya está bien escrita.
Los ejemplos deben ejecutarse.	El cuaderno genera reportes de split, preprocesado y evaluación por slices.
Las features también tienen tiempo.	Point-in-time correctness evita entrenar con información que aún no existía.
El split debe poder consultarse.	Una tabla materializada permite checks SQL, CI y revisión por pares.
Los backfills cambian la evidencia.	Si cambia el snapshot, quizá necesitas re-split o invalidar métricas anteriores.

Para saber más

Kapoor, S. y Narayanan, A. (2023). Leakage and the Reproducibility Crisis in Machine-Learning-Based Science. *Patterns*, 4(9), 100804. DOI

Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S. y Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020*. Paper

Wilson, E. B. (1927). Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association*, 22(158), 209-212. DOI

DVC. (2026). What is DVC?

MLflow. (2026). Dataset Tracking

MLflow. (2026). Evaluation Dataset Concepts

Apache Airflow. (2026). Data-aware scheduling

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377-387. DOI

Databricks. (2026). Point-in-time feature joins

Dagster. (2026). Testing assets with asset checks

dbt Labs. (2026). Add data tests to your DAG

DuckDB. (2026). CSV Loading

Feast. (2026). Point-in-time joins

Tecton. (2026). Construct Training Data

scikit-learn. (2026). GroupKFold

scikit-learn. (2026). Pipeline

scikit-learn. (2026). StratifiedGroupKFold

scikit-learn. (2026). TimeSeriesSplit

scikit-learn. (2026). train_test_split

scikit-learn. (2026). Common pitfalls and recommended practices

---

Notas

1. scikit-learn. (2026). *train_test_split*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
2. scikit-learn. (2026). *GroupKFold*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
3. scikit-learn. (2026). *StratifiedGroupKFold*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
4. scikit-learn. (2026). *TimeSeriesSplit*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
5. Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI. ↩
6. Kapoor, S. y Narayanan, A. (2023). Leakage and the Reproducibility Crisis in Machine-Learning-Based Science. *Patterns*, 4(9), 100804. DOI. ↩
7. Lewis, P. y otros (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020*. Paper. ↩
8. scikit-learn. (2026). *Pipeline*. https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html. Consultado el 22 de junio de 2026. ↩
9. scikit-learn. (2026). *Common pitfalls and recommended practices*. https://scikit-learn.org/stable/common_pitfalls.html. Consultado el 22 de junio de 2026. ↩
10. DVC. (2026). *What is DVC?* https://dvc.org/doc/user-guide/what-is-dvc. Consultado el 6 de junio de 2026. ↩
11. MLflow. (2026). *Dataset Tracking*. https://mlflow.org/docs/latest/ml/dataset/. Consultado el 22 de junio de 2026. ↩
12. MLflow. (2026). *Evaluation Dataset Concepts*. https://mlflow.org/docs/latest/genai/concepts/evaluation-datasets/. Consultado el 22 de junio de 2026. ↩
13. Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley. ↩
14. 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. ↩

.5. Databricks. (2026). *Point-in-time feature joins*. https://docs.databricks.com/aws/en/machine-learning/feature-store/time-series. Consultado el 22 de junio de 2026. ↩

.6. Tecton. (2026). *Construct Training Data*. https://docs.tecton.ai/docs/reading-feature-data/reading-feature-data-for-training/constructing-training-data. Consultado el 22 de junio de 2026. ↩

.7. Feast. (2026). *Point-in-time joins*. https://docs.feast.dev/getting-started/concepts/point-in-time-joins. Consultado el 22 de junio de 2026. ↩

.8. Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377-387. https://doi.org/10.1145/362384.362685 ↩

.9. dbt Labs. (2026). *Add data tests to your DAG*. https://docs.getdbt.com/docs/build/data-tests. Consultado el 22 de junio de 2026. ↩

!0. Dagster. (2026). *Testing assets with asset checks*. https://docs.dagster.io/guides/test/asset-checks. Consultado el 22 de junio de 2026. ↩

!1. DuckDB. (2026). *CSV Loading*. https://duckdb.org/docs/stable/data/csv/overview. Consultado el 22 de junio de 2026. ↩

!2. Apache Airflow. (2026). *Data-aware scheduling*. https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/datasets.html. Consultado el 22 de junio de 2026. ↩

!3. Wilson, E. B. (1927). Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association*, 22(158), 209-212. https://doi.org/10.1080/01621459.1927.10502953 ↩

!4. Efron, B. (1979). Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1), 1-26. https://doi.org/10.1214/aos/1176344552 ↩

Capítulo 04: Features y representaciones: de tablas a embeddings

Entrando en el tema

En el capítulo anterior congelamos una partición honesta. Ahora toca una pregunta igual de importante: ¿qué entra realmente al modelo? Un dataset no se convierte solo en inteligencia por estar en una tabla. Hay que transformar fechas, categorías, texto y metadatos en números que una función pueda operar.

Ese paso se llama representación. A veces es tan sencillo como convertir `email` en una columna binaria. A veces implica normalizar una fecha. A veces convierte un texto completo en un vector de cientos o miles de dimensiones. En todos los casos hay una decisión de ingeniería: qué información permitimos, qué información prohibimos, cuándo se ajusta la transformación y qué versión queda registrada.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Distinguir dato crudo, feature y representación.	No llamas “dato” a cualquier columna sin mirar cómo se transforma.
Diseñar un contrato de features.	Declaras columnas permitidas, prohibidas, tipos y `fit_scope`.
Explicar one-hot, normalización, TF-IDF y embeddings.	Puedes escribir sus fórmulas y decir qué aprende cada pieza.
Evitar leakage de representación.	Ajustas vocabulario, escalas e IDF solo con train.
Leer una dimensión de embedding como coste y contrato.	No tratas `d=64`, `d=768` o `d=3072` como decoración.
Evaluar una búsqueda vectorial mínima.	Miras top-k, términos fuera de vocabulario, metadata y slices.
Detectar feature skew.	Preguntas si offline y online calculan la misma feature con el mismo tiempo de observación.
Diferenciar proxy de métrica formal.	No llamas recall@k a una cobertura top-k sin relevancia anotada.

La idea común detrás de todos esos objetivos es que representar no es comprimir por comodidad. Representar es decidir qué señales entran, qué señales quedan fuera, cómo se ajustan las transformaciones y qué contrato garantiza que producción verá lo mismo que vimos al evaluar. Una feature puede parecer una columna inocente y ser un atajo; un embedding puede parecer semántica y ser una cercanía sin evidencia.

La frase central del capítulo:

> Una feature no es una columna: es una promesa sobre cómo una columna se convierte en señal.

La escena: el modelo no sabe leer tu tabla

Imagina una tabla con casos de soporte académico. Tiene columnas como `created_at`, `product`, `channel`, `student_id`, `source_id`, `label` y `text`. Una persona puede leerla y entender bastante: esto va de becas, aquello de matrícula, esto llegó por email, aquello por portal.

El modelo, en cambio, no ve “becas” como idea ni “email” como canal humano. Ve números. Si le das texto, alguien debe tokenizarlo o vectorizarlo. Si le das fechas, alguien debe convertirlas en una escala útil. Si le das categorías, alguien debe decidir si se codifican como one-hot, índices, embeddings aprendidos o metadata para filtrar.

Aquí aparece una parte muy de ingeniería de datos e IA: no basta con tener datos. Hay que construir una representación coherente, reproducible y alineada con el uso real. Si esa representación se construye mal, el modelo puede aprender una señal falsa, olvidar una señal importante o medir con información que no debería haber visto.

Qué no es una feature

Una feature no es cualquier columna que tengamos a mano. `student_id` existe en la tabla, pero quizá no debería entrar al modelo. Si entra, el sistema podría aprender estilos de estudiantes concretos en vez de patrones generales. `source_id` existe, pero puede actuar como atajo si cada documento está demasiado asociado a una etiqueta. `label` existe, pero usarla como entrada sería convertir la respuesta en pista.

Tampoco es una transformación hecha “porque mejora la métrica”. Si una feature mejora mucho el resultado, hay que preguntar por qué. Puede ser una señal legítima, como la antigüedad de un caso. Puede ser una fuga, como una fecha de resolución disponible solo después de cerrar el ticket. Puede ser una variable de identidad, como un código interno que en train coincide con la etiqueta pero en producción cambia.

Y un embedding no es conocimiento puro. Es una representación numérica aprendida o calculada. Puede capturar parecido semántico, pero no garantiza verdad, actualidad, permisos ni relación exacta. Si un vector store devuelve un fragmento cercano, todavía necesitamos comprobar si ese fragmento contiene evidencia suficiente para la pregunta.

Qué sí es una representación

Una representación es la forma en que describimos un objeto para que un algoritmo pueda trabajar con él. En aprendizaje estadístico, el punto de partida suele ser una matriz X, donde cada fila representa un ejemplo y cada columna representa una feature.[1] La etiqueta o valor que queremos predecir suele separarse como y.

$$X \in \mathbb{R}^{n \times d}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
X	Matriz de features.	24 casos por 49 features.
n	Número de ejemplos.	24 filas.
d	Número de features.	49 columnas generadas por el cuaderno.
$x_{i,j}$	Valor de la feature j para el ejemplo i.	`product__becas = 1`.

En palabras: X es la tabla ya convertida a números útiles para el modelo. No incluye todo lo que existe en el dataset; incluye solo las señales que el contrato permite usar como entrada.

En un proyecto de IA moderna, X puede mezclar varias familias de señal:

FAMILIA	EJEMPLO	QUÉ APORTA
Numérica	Días desde la primera fecha de train.	Orden temporal, recencia, duración.
Categórica	Producto, canal, país, tipo de usuario.	Segmentos y contexto operativo.
Texto disperso	TF-IDF, bolsa de palabras, BM25.	Coincidencia lexical y términos técnicos.
Texto denso	Embeddings de frases o documentos.	Parecido semántico y paráfrasis.
Metadata	`case_id`, permisos, versión de documento.	Trazabilidad, filtros y auditoría.

La clave es no mezclar funciones. Metadata puede ser imprescindible para auditar o filtrar, pero no necesariamente debe ser feature del modelo. El contrato decide esa frontera.

Familias de features: separar antes de transformar

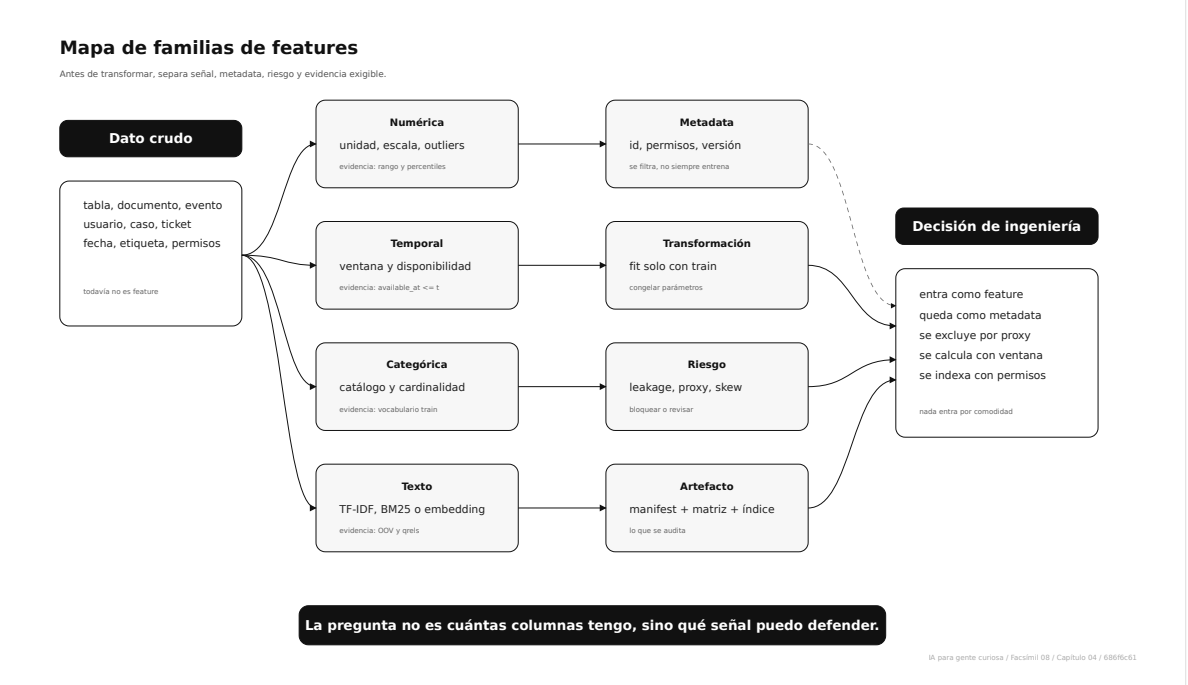
Una representación profesional empieza antes del código. Empieza clasificando qué tipo de señal tienes delante. No se transforma igual una fecha, una categoría con seis valores, una categoría con miles de valores, una descripción de texto, una etiqueta humana, un identificador técnico o una fecha de disponibilidad. Si todo entra en el mismo saco, el pipeline puede funcionar por accidente, pero será difícil de explicar, difícil de reproducir y peligroso de llevar a producción.

Para un ingeniero de datos, esta separación es una forma de proteger el sistema. Una feature numérica exige rango, unidad y política de nulos. Una feature temporal exige ventana y fecha de observación. Una feature categórica exige catálogo, tratamiento de desconocidos y decisión sobre cardinalidad. Un texto exige vocabulario, tokenización, idioma, normalización y quizá embeddings. La metadata exige otra pregunta: ¿sirve para filtrar y auditar, o realmente debe entrar como señal del modelo?

FAMILIA	EJEMPLO REALISTA	DECISIÓN TÉCNICA	RIESGO TÍPICO	EVIDENCIA QUE PEDIRÍA
Numérica	`importe_pendiente`, `dias_desde_ticket`	Escala, unidad, rango, outliers.	Que una magnitud grande domine sin aportar más información.	Histograma, percentiles, regla de nulos y transformación.
Temporal	`tickets_7d_as_of_created_at`	Ventana, granularidad y fecha de disponibilidad.	Usar datos que todavía no existían en el momento de decidir.	`event_timestamp`, `available_at` y test point-in-time.
Categórica baja	`channel`, `product`, `country`	One-hot, catálogo y desconocidos.	Crear columnas nuevas silenciosamente en producción.	Vocabulario ajustado con train y política `unknown`.
Categórica alta	`student_id`, `center_id`, `course_id`	Agrupar, excluir, usar metadata o embeddings aprendidos.	Memorizar identidades o crear miles de columnas poco útiles.	Cardinalidad, frecuencia mínima y revisión de proxy.
Texto lexical	`text`, `title`, `subject`	TF-IDF, BM25, stopwords, idioma y vocabulario.	Ajustar IDF con test o perder siglas y códigos.	Vocabulario train, OOV y baseline lexical.
Texto denso	Chunk, pregunta, caso completo	Encoder, dimensión, normalización, métrica.	Confundir cercanía semántica con evidencia válida.	Modelo, versión, dimensión, qrels y métricas de ranking.
Metadata	`case_id`, `source_id`, permisos, versión	Filtrar, auditar, unir y explicar.	Meter identificadores como atajo predictivo.	Lista de columnas prohibidas y uso explícito en filtros.

La alta cardinalidad merece atención propia. Un one-hot de `product` con seis categorías es razonable. Un one-hot de `student_id` con 200.000 valores casi nunca lo es: sube dimensión, memoria y riesgo de memorizar personas. Si el identificador es necesario para permisos, deduplicación o trazabilidad, se conserva como metadata. Si se quiere usar como señal, hay que justificar por qué no es un proxy peligroso y cómo se comporta cuando aparece una identidad nueva.

El *target encoding* (codificación por objetivo) es todavía más delicado. La idea es codificar una categoría usando una estadística del objetivo, por ejemplo la tasa histórica de reclamación de un centro. Puede ser útil en problemas tabulares, pero también puede filtrar la respuesta si se calcula con todo el dataset o si una categoría aparece muy pocas veces. La advertencia general coincide con la literatura sobre leakage en minería de datos: una transformación puede introducir información que no estaría disponible en el momento real de predicción.[2] Si se usa una codificación supervisada, debe calcularse dentro de cada fold o solo con train, suavizar categorías raras y documentar la política para categorías nuevas.



Separar familias de features evita que metadata, identidad, tiempo y texto acaben mezclados sin contrato.

Cómo se construye una feature por dentro

Construir una representación es una cadena de decisiones. Primero elegimos columnas permitidas. Después definimos transformaciones. Luego ajustamos lo que tenga que aprenderse usando solo train. Finalmente transformamos train, validation y test con los parámetros ya congelados.

Una feature numérica puede normalizarse con una escala min-max estándar. La idea es llevar un valor a una escala común usando los extremos observados en el conjunto con el que se ajusta la transformación. La documentación de `MinMaxScaler` expresa esta misma transformación para escalar cada feature dentro de un rango dado.[3] En un pipeline profesional, ese ajuste pertenece a train; scikit-learn insiste en esta separación porque ajustar preprocesado con datos reservados es una fuente clásica de fuga de información.[4]

$$\tilde{x} = \frac{x - x_{min}^{train}}{x_{max}^{train} - x_{min}^{train}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Valor original.	Día 10 desde el inicio.
x_{min}^{train}	Mínimo observado en train.	Día 0.
x_{max}^{train}	Máximo observado en train.	Día 15.
$\tilde{x}$	Valor normalizado.	$10/15 = 0.67$.

En palabras: el valor se reubica dentro de una escala común usando extremos aprendidos en train. Si validation o test participan en esos extremos, la evaluación deja de ser limpia.

La parte importante está en los superíndices: mínimo y máximo salen de train. Si usas todo el dataset, test participa en la escala. Puede parecer poco, pero en pipelines grandes esa pequeña comodidad se acumula con imputación, selección de variables, vocabularios e índices. Por eso las librerías maduras recomiendan encapsular preprocesado y modelo dentro de un `Pipeline`: no por estética, sino para que cada fold, split o experimento ajuste transformaciones en el sitio correcto.[5]

Una categoría puede codificarse con one-hot, una codificación que crea una columna binaria por categoría del vocabulario ajustado.[6]

$$onehot(c)_k = \begin{cases} 1 & \textbf{si } c = k \\ 0 & \textbf{si } c \neq k \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
c	Categoría de la fila.	`becas`.
k	Categoría concreta del vocabulario.	`matricula`.
$onehot(c)_k$	Columna binaria para k.	0 si el producto es `becas`.

En palabras: cada categoría congelada se convierte en una columna de sí/no. La categoría no se convierte en un número ordinal que invente distancia entre `becas` y `matricula`.

Si en train vemos `becas`, `horarios`, `matricula`, `pagos`, `practicas` y `titulos`, esas son las columnas que crea el contrato. Si mañana aparece `doctorado`, el sistema no debería improvisar silenciosamente una columna nueva en producción. Debe registrarlo como categoría desconocida, decidir cómo tratarla y actualizar el contrato si procede.

Para texto lexical, TF-IDF pondera términos usando frecuencia local y rareza global. Es una representación clásica en recuperación de información y clasificación textual; scikit-learn la implementa como transformación de recuentos a una matriz de features TF-IDF, con

referencias a Manning, Raghavan y Schütze.[7] Una forma habitual es:

$$tfidf(t,d) = tf(t,d) \cdot \left(\log \frac{1+N}{1+df(t)} + 1 \right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
t	Término.	`beca` .
d	Documento o texto de una fila.	“requisitos de beca”.
$tf(t,d)$	Veces que aparece t en d.	1.
N	Número de textos de train.	15.
$df(t)$	Número de textos de train que contienen t.	2.
$tfidf(t,d)$	Peso final del término.	Mayor si el término es informativo.

En palabras: TF-IDF sube términos frecuentes dentro de un texto, pero baja términos que aparecen en muchos textos. Por eso ayuda a rescatar palabras específicas sin confundirlas con vocabulario común del corpus.

BM25, una familia clásica de ranking lexical, parte de una intuición parecida pero ajusta saturación de frecuencia y longitud del documento.[8] En una formulación habitual, la puntuación de un documento D para una consulta Q se expresa así:

$$\text{BM25}(D,Q) = \sum_{q_i \in Q} IDF(q_i) \frac{f(q_i,D)(k_1+1)}{f(q_i,D) + k_1 \left(1 - b + b\frac{|D|}{avgdl}\right)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q_i	Término de la consulta.	`matricula`.
$f(q_i, D)$	Frecuencia del término en el documento.	Cuántas veces aparece `matricula`.
$IDF(q_i)$	Peso de rareza global del término.	Sube si aparece en pocos documentos.
$	D	$
$avgdl$	Longitud media del corpus.	Media de tokens por documento.
k_1, b	Parámetros de saturación y normalización por longitud.	Controlan cuánto premia repetición y documento largo.

En palabras: BM25 premia que el documento contenga términos de la consulta, pero no deja que repetir una palabra cien veces dispare la puntuación sin límite. También corrige el efecto de documentos más largos.

En RAG híbrido, estas señales lexicales siguen siendo muy útiles: los embeddings toleran paráfrasis; lo lexical rescata términos exactos, códigos, siglas y nombres propios. Si un alumno pregunta por `EXP-2026-41`, un embedding puede encontrar documentos parecidos, pero BM25 suele ser la primera defensa para no perder el identificador literal.

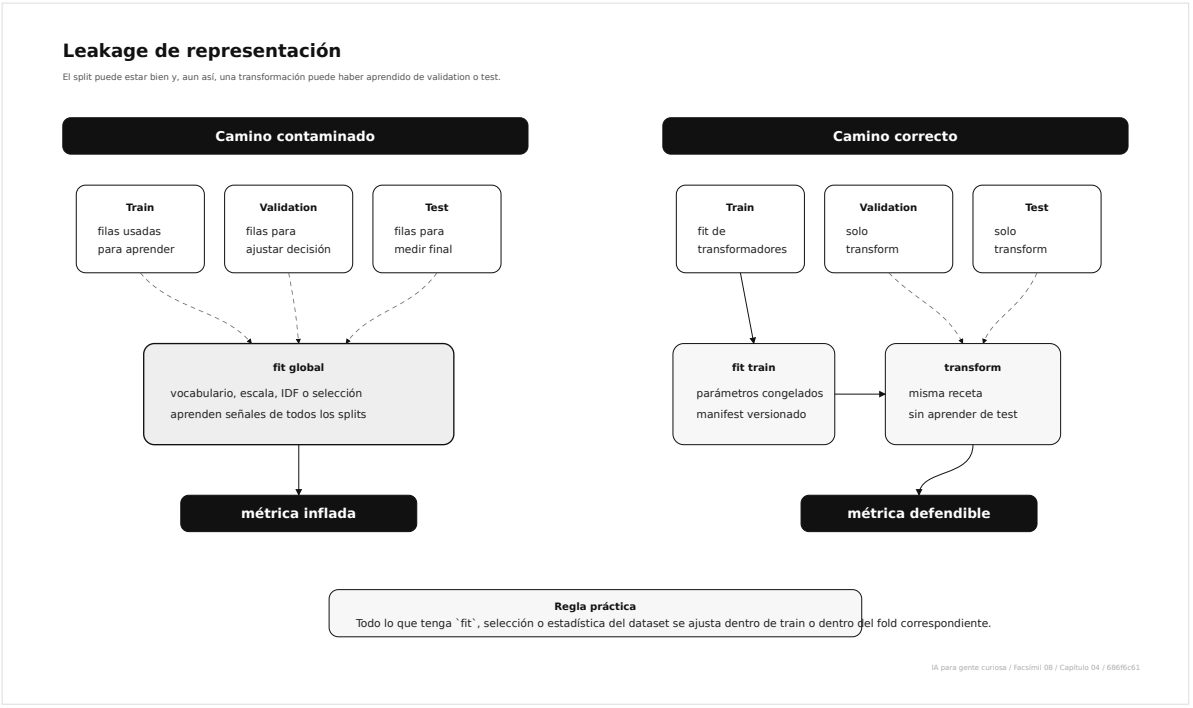
Leakage de representación: cuando la fuga no está en el split

En el capítulo 03 hablamos de splits y leakage. Aquí aparece una versión más sutil: el split puede estar bien, pero la transformación puede haber mirado donde no debía. Si calculas el vocabulario TF-IDF con todos los textos, validation y test ya influyeron en los pesos. Si eliges las mejores features mirando la métrica sobre todo el dataset, test ya participó en el diseño. Si normalizas con mínimos y máximos globales, los extremos de test modifican la escala de train. La fuga no siempre está en una columna llamada `label`; a veces está en una decisión cómoda de preprocesado.

Kaufman, Rosset, Perlich y Stitelman describen el leakage como información que no estaría disponible en el momento real de predicción y que, aun así, entra durante entrenamiento o evaluación.[9] En representación, esa información puede colarse por tres puertas: columnas posteriores al evento, transformaciones ajustadas fuera de train y selección de features hecha con el conjunto reservado.

Un caso muy realista: quieres predecir si una consulta de matrícula será prioritaria. Tienes `created_at`, `text`, `channel`, `status`, `resolved_at`, `agent_notes` y `label`. `resolved_at` y `agent_notes` son tentadores porque correlacionan mucho con el resultado. Pero si se rellenan

después de atender el caso, no son señales disponibles al decidir. Otro caso: haces target encoding de `center_id` usando toda la tabla. Si un centro aparece solo en test, acabas usando su tasa real de test para codificarlo. La métrica sube; el sistema no ha aprendido mejor, solo ha recibido una pista.



El leakage de representación aparece cuando un transformador aprende de filas que deberían servir solo para validar o medir.

Selección y reducción de features: menos columnas puede ser mejor ingeniería

Añadir features no es gratis. Aumenta memoria, tiempo de entrenamiento, riesgo de overfitting, coste de serving, dificultad de explicación y superficie de errores. En datos tabulares pequeños, diez features bien entendidas pueden ser más defendibles que quinientas columnas generadas sin criterio. En texto y embeddings, lo mismo: subir dimensión o vocabulario puede mejorar señales, pero también introduce ruido, coste y dificultad de evaluación.

La selección de features busca responder una pregunta concreta: ¿qué variables aportan señal fuera de la casualidad y del atajo? Hay técnicas filtradas, como mirar varianza, correlación o información mutua; técnicas embebidas, como regularización; técnicas wrapper, como evaluar subconjuntos; y técnicas de inspección posterior, como permutation importance. scikit-learn documenta permutation importance como una forma de medir cuánto empeora una métrica al permutar una feature, siempre sobre un conjunto de validación o test adecuado, no sobre los mismos datos usados para ajustar.[10]

La reducción de dimensionalidad responde a otro problema: representar muchos valores en menos coordenadas conservando estructura útil. PCA, SVD y otras técnicas aparecen en textos clásicos de aprendizaje estadístico como herramientas para proyectar datos a espacios de menor dimensión.[11] En este capítulo no necesitamos convertirlo en un curso de álgebra lineal, pero sí entender su contrato: si la proyección se ajusta con train, validation y test se transforman con esa proyección congelada. Si la ajustas con todo, vuelves a contaminar.

DECISIÓN	SIRVE PARA	CUIDADO PRINCIPAL	EJEMPLO ÚTIL
Eliminar varianza cero	Quitar columnas constantes.	Puede cambiar con producción si el catálogo crece.	Una categoría que nunca aparece en train.
Revisar correlación	Detectar duplicación fuerte entre features.	Correlación no implica causalidad ni utilidad.	`dias_abierto` y `horas_abierto`.
Permutation importance	Medir impacto de una feature en una métrica.	Debe hacerse con validación honesta.	Ver si `source_id` domina por atajo.
Regularización	Penalizar complejidad en modelos lineales.	Depende de escala y del modelo.	Muchas variables dispersas de texto.
PCA/SVD	Reducir dimensión conservando estructura.	Menos explicación directa por columna.	Vectores dispersos de alta dimensión.
Filtro por dominio	Bloquear proxies o variables post-evento.	Exige criterio humano y documentación.	Excluir `resolved_at` de un modelo de priorización.

La regla que me gusta usar es sencilla: si una feature no se puede explicar, versionar, calcular en el momento correcto y medir en validación honesta, todavía no está lista para producción. Puede quedarse en exploración, pero no en el contrato publicable.

Una selección de features bien hecha no busca adelgazar por estética. Busca reducir atajos, coste y fragilidad. Si una columna domina la predicción, el siguiente paso no es celebrarlo: es preguntarse si esa columna existirá en producción, si estaba disponible en el momento correcto, si es un proxy delicado y si la métrica sigue aguantando cuando la revisas por slices. Esa conversación es menos vistosa que subir un punto de accuracy, pero es mucho más cercana a cómo se evita deuda técnica en IA.

Embeddings: vectores densos con contrato

Un embedding es un vector denso. La notación siguiente es la forma habitual de escribir que un encoder parametrizado transforma un objeto en un vector dentro de un espacio de dimensión m, idea que aparece en modelos neuronales de lenguaje, Word2Vec, BERT y

Sentence-BERT.[12][13]

$$e = f_\theta(objeto) \in \mathbb{R}^m$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$objeto$	Lo que queremos representar.	Texto de un caso, imagen, usuario, producto.
f_θ	Encoder o función de representación.	Modelo de embeddings o encoder local.
θ	Parámetros del encoder.	Pesos entrenados o reglas congeladas.
m	Dimensión del vector.	64 en el cuaderno; 768 en muchos modelos BERT.
e	Vector resultante.	`[0.0, -0.12, ...]`.

En palabras: el encoder transforma un objeto en coordenadas. Esas coordenadas solo son comparables si query, documentos y evaluación usan el mismo encoder, versión, dimensión y normalización.

Para un ingeniero, la dimensión m no es un número decorativo. Afecta memoria, coste de indexación, latencia de búsqueda, tamaño del índice, elección de base vectorial y facilidad para mantener varias versiones. También afecta a la evaluación: cambiar de encoder o de dimensión puede reordenar vecinos aunque el corpus sea el mismo. Por eso un embedding debe venir con ficha técnica igual que un dataset: modelo, versión, dimensión, normalización, métrica de similitud, fecha de corte y uso permitido.

La historia moderna de embeddings no empieza con los LLM actuales. Bengio y colaboradores ya propusieron representar palabras mediante vectores aprendidos dentro de modelos de lenguaje neuronales.[14] Word2Vec popularizó embeddings preentrenados eficientes para palabras.[15] BERT llevó la idea a representaciones contextuales: el vector de una palabra depende de la frase completa.[16] Sentence-BERT adaptó arquitecturas BERT para producir embeddings de frases útiles en similitud semántica.[17]

La similitud coseno compara direcciones y es una medida clásica en el modelo vectorial de recuperación de información.[18]

$$\cos(a, b) = \frac{a \cdot b}{\|a\|\|b\|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a, b	Dos vectores.	Query y caso indexado.
$a \cdot b$	Producto escalar.	Suma de productos dimensión a dimensión.
$\|a\|$	Norma de a.	Longitud del vector.
$\cos(a, b)$	Similitud por dirección.	1 si apuntan igual; 0 si son ortogonales.

En palabras: el coseno compara orientación más que tamaño. Dos vectores pueden ser cercanos si apuntan en dirección parecida, aunque sus magnitudes originales fueran distintas.

Si el sistema normaliza los embeddings a norma 1 antes de guardarlos, el cálculo se simplifica: comparar por coseno queda muy cerca de comparar por producto escalar. La normalización L2 es una operación estándar de preprocesado vectorial.[19] Se escribe así:

$$\hat{e} = \frac{e}{\|e\|_2}$$

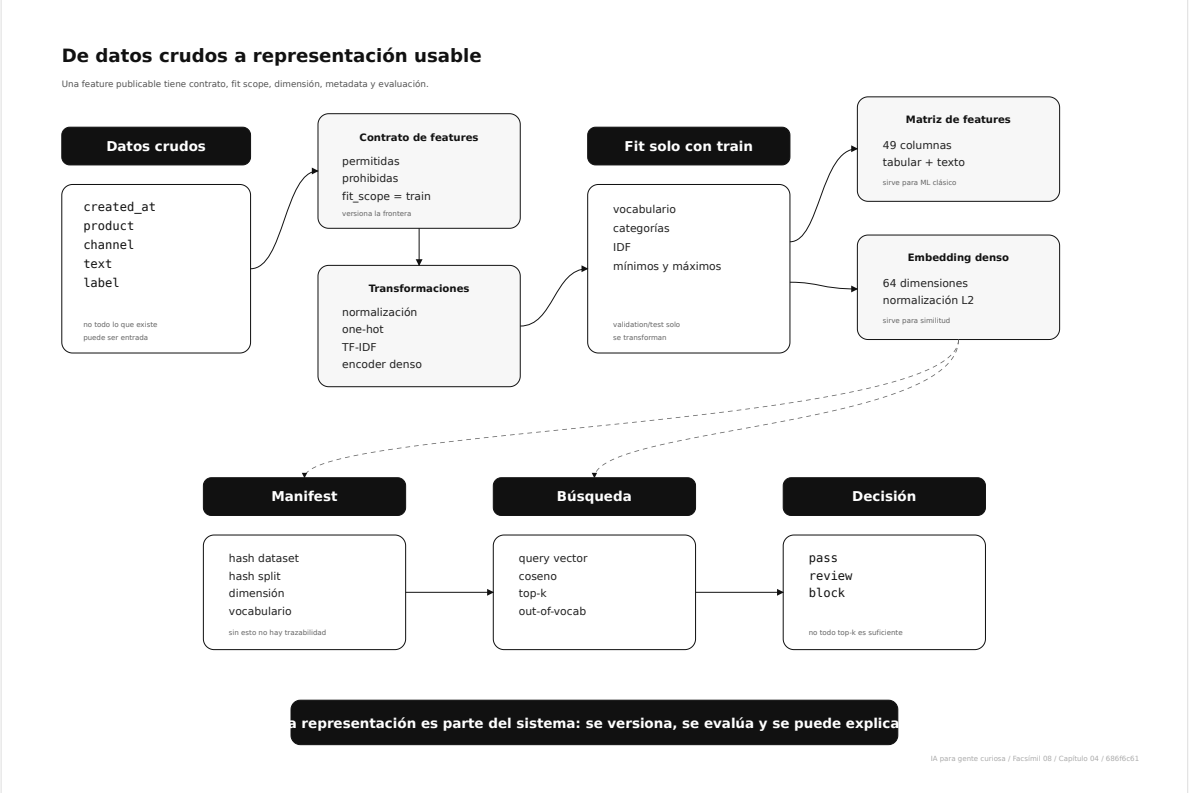
SÍMBOLO	SIGNIFICADO	EJEMPLO
e	Vector original.	Embedding antes de indexar.
$\|e\|_2$	Norma euclídea del vector.	Longitud geométrica.
$\hat{e}$	Vector normalizado.	El que se guarda si el índice espera norma 1.

En palabras: normalizar divide el vector por su longitud para dejarlo con norma 1. No añade conocimiento; prepara la geometría para comparar de forma coherente.

Esta fórmula no dice que el embedding sea “mejor”. Dice que lo dejas en una geometría consistente para que la comparación sea estable. En producción conviene registrar si el índice usa coseno, producto escalar o distancia euclídea, porque cambiar la métrica sin reindexar y reevaluar puede alterar el ranking aunque el texto sea el mismo.

Dimensión no significa “inteligencia”. Significa longitud del vector y, por tanto, memoria, coste de cálculo, coste de índice y capacidad de representar matices. Un embedding de 64 dimensiones cabe barato y sirve como baseline local. Uno de 768 o más puede capturar mucha más estructura, pero cuesta más guardar, comparar y servir.

La anatomía de una representación publicable



Una representación publicable conecta contrato, transformación, fit con train, manifiesto y evaluación.

Evaluar representaciones de recuperación

Una búsqueda vectorial no se valida mirando si “parece razonable”. Necesitas consultas, documentos relevantes anotados y una métrica que responda a la decisión real. En recuperación de información, si $Rel(q)$ es el conjunto de documentos relevantes para la consulta q, y $Ret_k(q)$ son los k documentos recuperados por el sistema, dos métricas básicas son precision@k y recall@k.[20]

$$\text{Precision@k}(q) = \frac{|Rel(q) \cap Ret_k(q)|}{k}$$

$$\text{Recall@k}(q) = \frac{|Rel(q) \cap Ret_k(q)|}{|Rel(q)|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Consulta evaluada.	“justificante de pago pendiente”.
$Rel(q)$	Conjunto anotado de documentos relevantes.	Casos o chunks que deberían responder a esa consulta.
$Ret_k(q)$	Resultados devueltos entre las primeras k posiciones.	Top 3 del índice.
Precision@k	Qué proporción del top-k es relevante.	Evita llenar contexto de ruido.
Recall@k	Qué proporción de lo relevante apareció en top-k.	Evita dejar fuera evidencia necesaria.

En palabras: precision@k mira la limpieza de los primeros resultados; recall@k mira cuánta evidencia relevante has conseguido traer. Si $k = 3$, recuperas tres documentos y solo uno está anotado como relevante, precision@3 vale $1/3$. Si para esa consulta había cuatro documentos relevantes y solo recuperaste uno, recall@3 vale $1/4$. No es una métrica decorativa: en RAG, esos tres documentos suelen ser los que pasan al prompt, al reranker o a la revisión humana.

Pero precision y recall no dicen dónde aparece el primer resultado bueno. Para eso se usa MRR, *Mean Reciprocal Rank*, una métrica clásica en recuperación de información cuando importa mucho que el primer resultado relevante aparezca pronto.[21]

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Q	Conjunto de consultas evaluadas.	30 preguntas reales de soporte.
q	Una consulta concreta.	“justificante de pago pendiente”.
$rank_q$	Posición del primer resultado relevante para q.	1 si el primero ya vale; 3 si aparece tercero.
MRR	Media del recíproco de esas posiciones.	Penaliza que lo relevante aparezca tarde.

En palabras: si el primer resultado relevante aparece en posición 1, aporta 1. Si aparece en posición 2, aporta 0,5. Si aparece en posición 5, aporta 0,2. Es muy útil para buscadores internos, asistentes de soporte y sistemas donde el usuario o el modelo suelen mirar pocos resultados.

Cuando no todas las evidencias relevantes valen lo mismo, necesitamos grados. Un fragmento puede responder exactamente, otro puede ser parcialmente útil y otro puede estar relacionado pero no resolver la pregunta. nDCG@k, propuesto dentro de la evaluación basada en ganancia acumulada por Järvelin y Kekäläinen, mide ranking con relevancia graduada y descuenta posiciones bajas.[22]

$$\mathrm{DCG@k}(q) = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

$$\mathrm{nDCG@k}(q) = \frac{\mathrm{DCG@k}(q)}{\mathrm{IDCG@k}(q)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
i	Posición del resultado dentro del ranking.	1, 2, 3...
rel_i	Grado de relevancia del resultado en posición i.	0 irrelevante, 1 parcial, 2 relevante, 3 excelente.
$\mathrm{DCG@k}$	Ganancia acumulada descontada hasta k.	Premia relevancia arriba.
$\mathrm{IDCG@k}$	Ganancia ideal si ordenaras perfecto.	El mejor ranking posible con esos juicios.
$\mathrm{nDCG@k}$	DCG normalizado entre 0 y 1.	Permite comparar consultas con distinta dificultad.

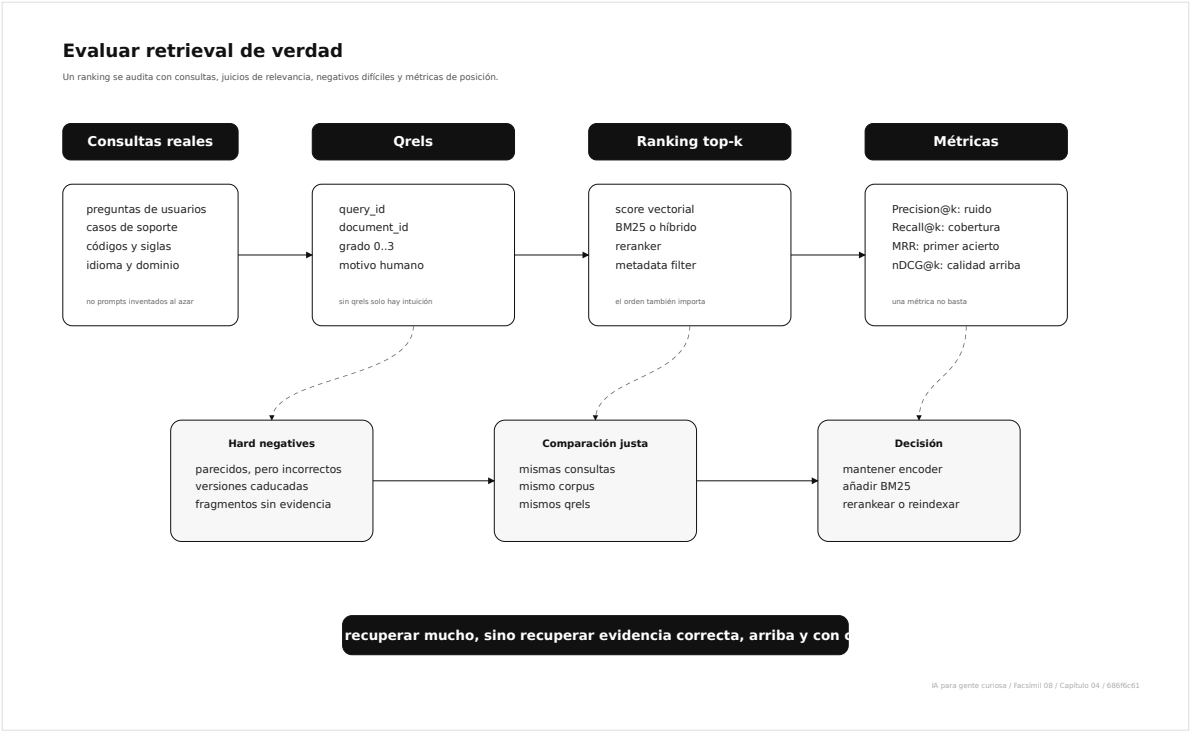
En palabras: nDCG@k castiga dos cosas a la vez: traer documentos poco relevantes y poner documentos buenos demasiado abajo. Para RAG es una métrica especialmente útil porque no basta con “algún chunk relevante en top-k”: quieres que los chunks más fuertes aparezcan arriba, antes de llenar contexto con ruido.

La diferencia no es académica de salón. Si haces un asistente de normativa universitaria, precision@k bajo significa que metes fragmentos irrelevantes en el prompt y aumentas el riesgo de respuesta confusa. Recall@k bajo significa que existe evidencia importante, pero el sistema no la trae. En un RAG de soporte, ambos errores se sienten: o respondes con ruido, o respondes sin la pieza que hacía falta.

El cuaderno de este capítulo trae un `qrels` mínimo, no una evaluación industrial completa. Eso significa que ya no miramos solo si el producto esperado aparece en el top-k: anotamos qué casos son relevantes para cada consulta, con qué grado y por qué. La cobertura por producto sigue siendo una señal barata para detectar errores groseros, pero precision@k, recall@k, MRR y nDCG@k se calculan con relevancia anotada. Para cerrar un sistema real

habría que ampliar ese `qrels` con más consultas, doble revisión, desacuerdos entre revisores, *hard negatives* y slices de dominio. Aun así, el mecanismo ya permite comparar encoder local, BM25, búsqueda híbrida, reranker y cambios de chunking sin discutir a ojo.

Un *hard negative* es un documento que se parece mucho a la respuesta correcta, pero no sirve para contestar. Por ejemplo, una política de becas del curso anterior puede ser semánticamente cercana a la consulta actual, pero estar caducada. Si el sistema la recupera arriba, el embedding no está “fallando de forma absurda”; está mostrando que la evaluación necesita distinguir parecido de evidencia válida. En sistemas de IA aplicada, esos casos son oro: obligan a introducir fecha, versión, permisos, reranking o reglas de filtrado.



> Una evaluación útil combina consultas reales, qrels, negativos difíciles y métricas que miran limpieza, cobertura y posición.

De quién viene cada fórmula y por qué está aquí

Hay una costumbre peligrosa en libros técnicos: poner fórmulas para que el texto parezca más serio. Aquí queremos justo lo contrario. Una fórmula entra solo si ayuda a tomar una decisión de ingeniería: cómo representar, cómo evitar leakage, cómo indexar, cómo comparar o cómo medir.

También conviene ser honestos con la palabra “de quién”. Algunas fórmulas tienen autores o familias muy reconocibles, como BM25. Otras no pertenecen a una persona concreta, sino a una tradición matemática o de ingeniería muy asentada, como escribir una matriz de datos

como $X \in \mathbb{R}^{n \times d}$ o normalizar un vector por su norma. En esos casos no buscamos un “inventor” decorativo: buscamos la referencia académica o técnica que justifica su uso.

FÓRMULA	DE DÓNDE VIENE	POR QUÉ SE USA AQUÍ	DECISIÓN DE INGENIERÍA QUE PERMITE
$X \in \mathbb{R}^{n \times d}$	Notación estándar de aprendizaje estadístico; Hastie, Tibshirani y Friedman la usan para razonar sobre matrices de datos, features y modelos.	Sitúa el capítulo: una tabla se convierte en matriz y cada columna pasa a tener significado operacional.	Separar filas, features, target y metadata antes de entrenar o evaluar.
Escalado min-max	Transformación clásica de preprocesado; scikit-learn la documenta en `MinMaxScaler`.	Explica por qué una feature numérica no debería entrar “tal cual” si su escala domina a otras.	Ajustar mínimos y máximos solo con train y evitar que test participe en el preprocesado.
One-hot encoding	Codificación categórica estándar; scikit-learn la documenta en `OneHotEncoder`.	Enseña que una categoría no es un número ordinal salvo que el dominio lo justifique.	Congelar vocabularios de categorías y decidir qué hacer con valores nuevos en producción.
TF-IDF	Recuperación de información y vectorización textual; Manning, Raghavan y Schütze la explican, y scikit-learn la implementa en `TfidfTransformer`.	Muestra cómo un texto puede pasar a features dispersas sin usar todavía un embedding neural.	Ajustar vocabulario e IDF solo con train; detectar términos fuera de vocabulario.
BM25	Robertson y Zaragoza, dentro del marco probabilístico de recuperación de información.	Da una base lexical fuerte para búsqueda y RAG híbrido, especialmente con códigos, siglas y nombres propios.	Comparar búsqueda lexical, vectorial e híbrida antes de elegir un índice.
$e = f_\theta(objeto) \in \mathbb{R}^m$	Notación habitual para encoders parametrizados; conecta con modelos neuronales de lenguaje, Word2Vec, BERT y Sentence-BERT.	Evita tratar “embedding” como magia: es una función que transforma un objeto en un vector.	Registrar modelo, versión, dimensión, normalización y tipo de objeto embebido.
Similitud coseno	Modelo vectorial de recuperación de información; Manning, Raghavan y Schütze la presentan como forma de comparar vectores.	Permite explicar por qué dos textos pueden estar cerca por dirección aunque no tengan la misma longitud.	Elegir métrica de índice y comprobar si query y documentos usan el mismo espacio vectorial.
Normalización L2	Operación estándar de álgebra lineal y preprocesado vectorial; scikit-learn la documenta en `Normalizer`.	Explica por qué muchos sistemas guardan embeddings con norma 1 antes de comparar.	Saber cuándo coseno y producto escalar son comparables y cuándo reindexar al cambiar métrica.

FÓRMULA	DE DÓNDE VIENE	POR QUÉ SE USA AQUÍ	DECISIÓN DE INGENIERÍA QUE PERMITE
$\mathrm{Precision@k}$	Métrica clásica de recuperación de información; Manning, Raghavan y Schütze.	Mide cuánto ruido metes en los primeros resultados que luego verá el modelo o el usuario.	Decidir si el top-k tiene suficiente limpieza para alimentar un RAG.
$\mathrm{Recall@k}$	Métrica clásica de recuperación de información; Manning, Raghavan y Schütze.	Mide si el sistema trae suficiente evidencia relevante entre los primeros k resultados.	Decidir si hace falta cambiar encoder, chunking, BM25, reranker o anotación de relevancia.
MRR	Métrica clásica de ranking usada en recuperación de información; Manning, Raghavan y Schütze la tratan dentro de evaluación de sistemas IR.	Mide cuánto tarda en aparecer el primer resultado relevante.	Decidir si el sistema sirve cuando el usuario o el LLM mira solo los primeros resultados.
$\mathrm{nDCG@k}$	Evaluación por ganancia acumulada descontada; Järvelin y Kekäläinen la formalizan para relevancia graduada.	Mide si los documentos más útiles aparecen arriba, no solo dentro del top-k.	Comparar encoders, BM25, híbridos, rerankers o cuantización cuando hay grados de relevancia.

La lectura práctica es esta: las fórmulas no están para memorizar símbolos. Están para que, cuando alguien proponga “metamos embeddings y ya”, puedas preguntar con precisión: ¿qué matriz estamos creando?, ¿qué se ajustó con train?, ¿qué encoder produjo el vector?, ¿qué dimensión tiene?, ¿qué métrica usa el índice?, ¿qué documentos eran relevantes y qué parte del top-k los recuperó?

Herramientas externas sin perder criterio

La forma superficial de leer este tema es quedarse con una frase tipo “usa un vector database” o “usa embeddings”. La forma de ingeniería es separar capas. Primero está el **preprocesado reproducible**: qué columnas entran, cómo se transforman y dónde se ajustan los parámetros. Después está el **contrato de datos**: qué tipo, rango, null, categoría o formato se acepta. Luego aparece la **capa de features** si hay que reutilizar definiciones offline y online. Y, si hay texto, documentos o chunks, aparece la **capa de embeddings e índice vectorial**. Cada herramienta externa debería entrar en una de esas capas, no en todas a la vez.

Para tabular clásico, scikit-learn sigue siendo una base muy buena porque obliga a pensar en `Pipeline` y transformadores por columna. `ColumnTransformer`, por ejemplo, permite aplicar normalización a numéricas, one-hot a categóricas y TF-IDF a texto, concatenando después todo en una sola matriz de features.[23] La pregunta de ingeniería no es “¿uso scikit-learn?”, sino “¿mi preprocesado queda dentro del pipeline o está repartido en celdas de notebook que nadie podrá reproducir?”.

Para contratos de datos, herramientas como Pandera o Great Expectations ayudan a declarar expectativas sobre tablas: tipos, columnas requeridas, rangos, nulls, categorías o reglas de negocio.[24][25] No sustituyen entender el dominio. Si declaras mal el contrato, la herramienta validará basura perfectamente. Su valor está en convertir una conversación difusa en una barrera ejecutable: “esta feature no entra si falta fecha de observación”, “esta categoría nueva bloquea”, “este porcentaje de nulos exige revisión”.

Cuando una feature debe existir tanto en entrenamiento como en producción, el problema cambia. Ya no basta con un pipeline local: necesitas una disciplina de feature store. Feast y Tecton son dos referencias útiles para entender esa arquitectura: definiciones versionadas, fuentes offline, serving online, entidades, ventanas temporales y joins point-in-time.[26][27] En un proyecto pequeño quizá no necesitas desplegar una plataforma completa; pero sí necesitas copiar la disciplina: definición única, timestamp de disponibilidad, manifest, tests y comparación offline/online.

En entornos cloud, la misma idea aparece integrada en plataformas de datos. Databricks documenta feature tables gobernadas en Unity Catalog, con claves primarias, namespace de catálogo, esquema y tabla; Google describe Vertex AI Feature Store como una capa de metadata y serving online sobre fuentes como BigQuery.[28][29] La decisión aquí no es “montar cloud porque sí”. Es preguntar si tu equipo ya vive en un lakehouse, warehouse o plataforma gestionada, y si te conviene aprovechar permisos, linaje, catálogo y serving que ya existen.

Con embeddings ocurre otra tentación: elegir proveedor por moda. OpenAI documenta los embeddings como vectores de números reales donde la distancia entre vectores mide relación entre textos, y permite trabajar con dimensión como parámetro en modelos compatibles.[30] Eso no te libera de evaluar. Un embedding más grande puede mejorar algunas tareas, pero también sube coste de almacenamiento, latencia y tamaño del índice. En ingeniería se registra siempre: modelo, versión, dimensión, normalización, fecha de generación, corpus usado, idioma, política de reindexado y métrica de evaluación.

Para guardar y consultar vectores, la elección depende más de tu sistema que del brillo de la herramienta. Si ya tienes Postgres y el caso es pequeño o mediano, `pgvector` puede ser suficiente porque guarda vectores junto con datos relacionales, joins, backups y permisos existentes.[31] Si necesitas un motor especializado con colecciones, payloads, filtros, cuantización o más control de búsqueda, Qdrant, Weaviate, Milvus o Pinecone entran en la conversación.[32][33][34][35]

La comparación útil no es “cuál es mejor”, sino “qué decisión protege”. Qdrant habla en términos de colecciones, vectores y payloads; eso te obliga a pensar en dimensión, métrica y metadata. Weaviate hace explícita la búsqueda híbrida, mezclando keyword y vector, útil cuando necesitas que siglas, códigos y nombres propios no desaparezcan detrás de la semántica. Milvus pone el foco en índices, métricas y balance entre rendimiento y corrección de la búsqueda. Pinecone insiste en metadata y filtrado porque muchos sistemas reales no pueden buscar “en todo”: tienen permisos, tenant, fecha, producto, país o tipo documental.

CAPA	HERRAMIENTAS POSIBLES	ÚSALAS CUANDO...	QUÉ EXIGIR ANTES DE CONFIAR
Preprocesado reproducible	scikit-learn `Pipeline`, `ColumnTransformer`	Tienes numéricas, categorías y texto en una misma matriz.	El `fit` ocurre solo con train y el pipeline se serializa o versiona.
Contrato de datos	Pandera, Great Expectations	Necesitas bloquear columnas, tipos, nulls, rangos o categorías inválidas.	Las reglas vienen del dominio y fallan con ejemplos negativos reales.
Feature store	Feast, Tecton, Databricks Feature Engineering, Vertex AI Feature Store	La misma feature vive offline y online, o necesitas joins temporales.	Hay `event_timestamp`, `created_timestamp`, point-in-time join y comparación offline/online.
Embeddings	OpenAI, Voyage, Cohere, Gemini, modelos open weights	El objeto es texto, chunk, imagen o consulta semántica.	Quedan registrados modelo, dimensión, normalización, coste, idioma, corpus y fecha de generación.
Índice vectorial	pgvector, Qdrant, Weaviate, Milvus, Pinecone, FAISS	Necesitas top-k, filtros, metadata, reindexado o búsqueda aproximada.	Se mide precision@k, recall@k, MRR y nDCG@k con qrels, y se prueban filtros de permisos.
Híbrido y reranking	BM25 + vector, Weaviate hybrid, rerankers externos	Hay códigos, siglas, nombres propios o consultas largas.	Se compara contra baseline lexical, vectorial e híbrido con las mismas consultas.

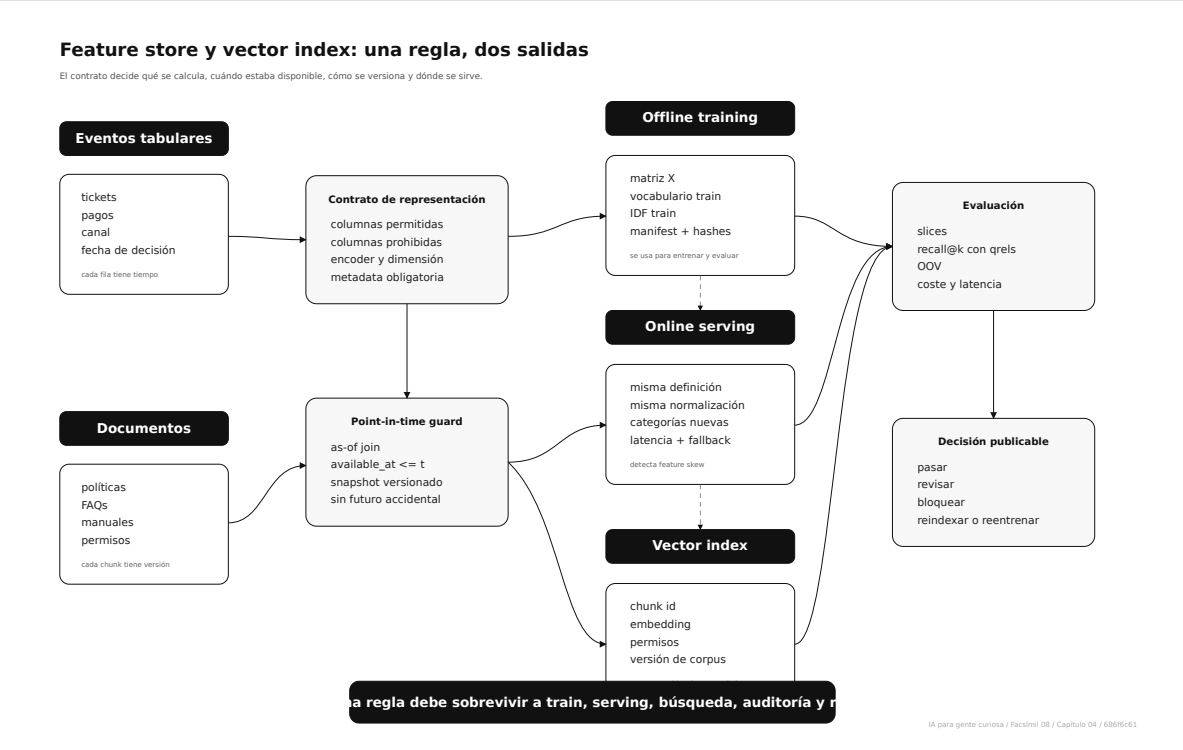
Un ejemplo concreto: tienes un asistente para normativa universitaria. Para la parte tabular de tickets usaría `Pipeline` y contratos de datos. Para features de producción, antes de montar una plataforma, exigiría una tabla con `entity_id`, `event_timestamp`, `available_at` y definición versionada. Para documentos, generaría embeddings con manifest, guardaría `document_id`, `chunk_id`, `version`, `permissions`, `created_at` y `embedding_model`, y empezaría con pgvector si el volumen cabe en Postgres. Solo saltaría a una base vectorial dedicada si necesito filtros complejos a escala, multi-tenant, cuantización, throughput alto o administración específica de índices. Y antes de celebrar nada, construiría un qrels de 30-50 consultas reales: ahí se ve si el sistema recupera evidencia o solo devuelve vecinos bonitos.

Paridad offline/online: el enemigo silencioso

Una de las formas más serias de romper un sistema de features es entrenar con una receta y servir con otra. En entrenamiento calculas `casos_abiertos_ultimos_7_dias` con una query sobre un snapshot histórico; en producción la calculas con una API que tiene retraso, redondea fechas de otra manera o incluye estados que el entrenamiento no veía. La métrica offline puede ser buena y, aun así, el sistema en vivo comportarse distinto. A esta diferencia se la suele llamar *training-serving skew* o *feature skew*.

El otro problema es temporal. Una feature puede ser correcta como cálculo y ser incorrecta como evidencia. Si un ticket se decide el 10 de junio, no puedes usar una columna que se rellenó el 12 de junio. Para evitarlo se usa *point-in-time correctness*: cada fila de entrenamiento solo puede unirse con valores de feature disponibles en el momento de decisión de esa fila. Feast lo documenta como *point-in-time joins*; Tecton lo trata como parte de la construcción de datos de entrenamiento para evitar que features futuras contaminen el dataset.[3637]

Un ejemplo cercano: quieres priorizar incidencias académicas. Podrías crear una feature `pagos_pendientes_al_crear_ticket`. Si la calculas mirando la base de pagos actual, quizá estás usando pagos resueltos después del ticket. La versión correcta necesita una fecha de observación: “qué sabía el sistema cuando se creó el ticket”. Esa frase parece burocrática, pero separa un modelo honesto de un modelo que aprende el futuro.



Un contrato de representación no termina en una matriz: también gobierna serving online, índice vectorial, evaluación y decisión de publicación.

Anatomía de un índice vectorial: no es una carpeta de embeddings

Un índice vectorial es una pieza de ingeniería, no un disco donde tiramos arrays. Recibe vectores, metadata, una métrica de distancia y una política de búsqueda. Devuelve vecinos aproximados o exactos, normalmente con filtros de payload, tenant, permisos o fecha. Si lo usamos en RAG, ese ranking puede decidir qué evidencia entra en el prompt. Por eso la pregunta correcta no es “¿qué vector database está de moda?”, sino “¿qué contrato de recuperación puedo medir y operar?”.

En volúmenes pequeños, una búsqueda exacta puede comparar la consulta contra todos los vectores. En volúmenes grandes, eso se vuelve caro. FAISS popularizó infraestructuras de búsqueda de similitud a gran escala, especialmente en GPU, y HNSW se convirtió en una familia muy usada de índices aproximados basados en grafos jerárquicos.[38][39] Aproximado significa esto: a cambio de velocidad y memoria razonable, mides si sigues recuperando los vecinos que importan.

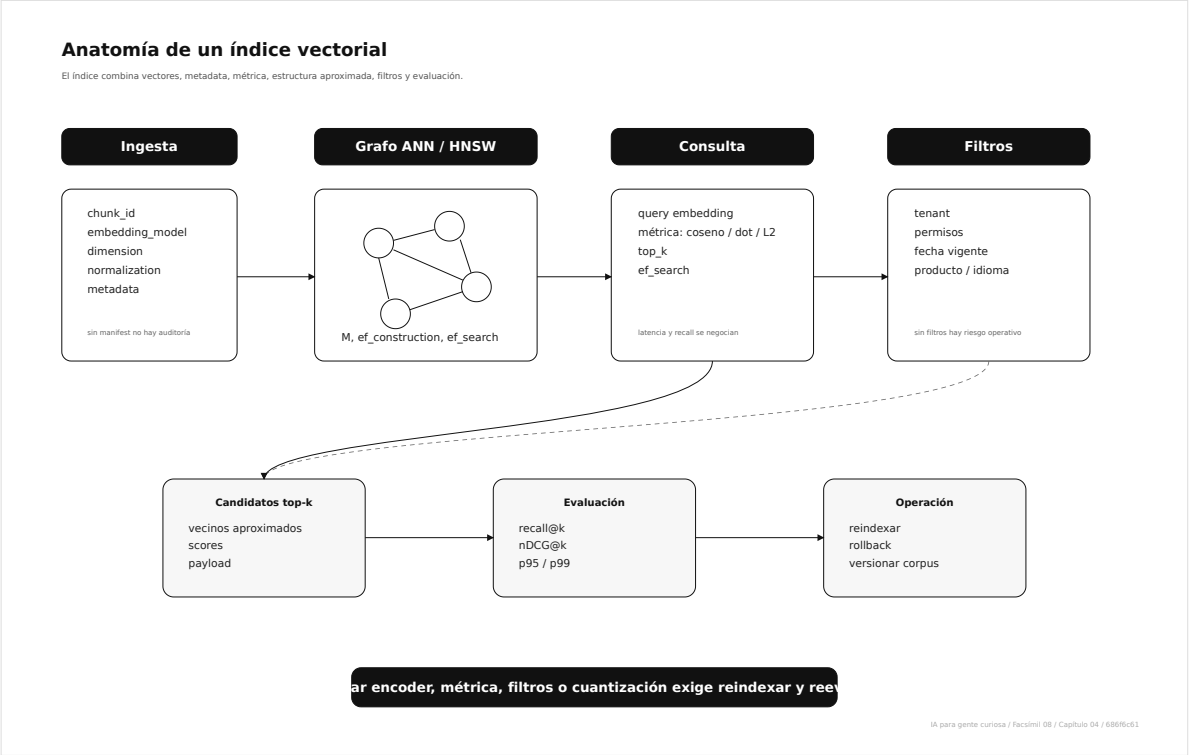
HNSW suele exponerse con parámetros que un ingeniero debe entender. `M` controla cuántas conexiones mantiene cada nodo del grafo; subirlo suele aumentar memoria y calidad de búsqueda. `ef_construction` afecta la calidad y coste de construir el índice. `ef_search` afecta el trabajo en consulta: subirlo puede mejorar recall, pero también latencia. Qdrant documenta estos parámetros dentro de su configuración de indexado, y Milvus también separa tipo de índice, métrica y parámetros de búsqueda.[40][41]

También aparece la compresión. Product Quantization, propuesta por Jégou, Douze y Schmid para búsqueda de vecinos cercanos, divide vectores en subespacios y los cuantiza para reducir memoria y acelerar búsqueda aproximada.[42] Es una herramienta potente, pero no gratuita: al comprimir puedes perder calidad de ranking. De nuevo, no se decide por intuición; se mide con qrels, slices, latencia y coste.

Los filtros de metadata son la otra mitad del sistema. Un índice que recupera un documento perfecto pero sin respetar permisos no es un éxito técnico; es un incidente. Pinecone, Qdrant, Milvus y Weaviate documentan filtrado por metadata o búsqueda híbrida porque en aplicaciones reales se busca dentro de subconjuntos: usuario, organización, idioma, fecha, producto, país, versión documental, estado legal o permisos.[43][44]

Cuando combinas ranking lexical y ranking vectorial, hay varias estrategias. Puedes concatenar candidatos, ponderar scores si están calibrados, usar un reranker o fusionar posiciones. Reciprocal Rank Fusion, de Cormack, Clarke y Buettcher, es una técnica clásica para fusionar rankings usando la posición de cada documento en distintas listas.[45] No hace falta que lo implementes en el primer día, pero sí conviene entender la idea: BM25 puede capturar códigos exactos; el embedding puede capturar paráfrasis; el reranker puede reordenar con más contexto. El sistema serio mide cada capa.

PIEZA DEL ÍNDICE	QUÉ CONTROLA	QUÉ PUEDE ROMPER	QUÉ MEDIRÍA
Dimensión	Memoria por vector y coste de distancia.	Latencia y tamaño de índice.	Memoria, p95/p99, recall@k.
Métrica	Coseno, producto escalar o L2.	Rankings distintos si no reindexas.	Comparación contra qrels.
HNSW `M`	Conectividad del grafo.	Memoria alta o búsqueda pobre.	Recall@k frente a memoria.
HNSW `ef_search`	Trabajo en consulta.	Latencia alta o vecinos perdidos.	Curva recall-latencia.
Cuantización	Compresión de vectores.	Pérdida de ranking fino.	nDCG@k antes/después.
Metadata filters	Alcance permitido de búsqueda.	Recuperar evidencia prohibida o caducada.	Tests por tenant, permisos y fecha.
Reindexado	Cuándo reconstruyes índice.	Mezclar modelos o versiones de corpus.	Manifest de versión y regresión de retrieval.



Un índice vectorial serio tiene parámetros, filtros, versiones, métricas y operación; no es solo una lista de embeddings.

En el día a día

En un equipo profesional, las features no deberían vivir escondidas dentro de un notebook. Deben tener nombre, definición, propietario, tipo, fuente, ventana temporal, reglas de null, método de cálculo, versión y criterio de retirada. Cuando la misma feature se usa para entrenamiento offline y predicción online, además aparece un problema clásico: que el cálculo de entrenamiento y el cálculo de producción no coincidan.

Ahí entra la idea de feature store: un sistema o una disciplina para compartir definiciones de features, materializarlas, servirlas y versionarlas. Feast, por ejemplo, documenta esta filosofía de definir entidades, feature views y fuentes para servir features de manera consistente.[46] Lo importante para este facsímil no es casarnos con una herramienta concreta, sino entender la responsabilidad: si una feature decide, esa feature debe poder explicarse.

En RAG ocurre algo parecido. El embedding de un chunk no es una columna tradicional, pero es un dato derivado. Debe conservar `document_id`, versión del documento, versión del chunking, encoder, dimensión, fecha de creación y permisos. Si cambias cualquiera de esas piezas, el índice ya no es el mismo aunque el texto visible parezca igual.

Por qué debería importarte

Una mala representación puede arruinar un sistema sin que el error sea evidente. El modelo compila, la API responde y la métrica puede subir, pero quizá aprendió una identidad, una fecha posterior, una categoría mal codificada o un embedding sin metadata. La representación es el lugar donde se decide qué puede aprender el sistema.

También es una cuestión de coste. Si tienes 10 millones de documentos y pasas de 384 a 3072 dimensiones, multiplicas almacenamiento y cálculo. Si además replicas el índice, guardas metadata y añades estructuras de vecinos aproximados, el coste ya no es marginal. FAISS y HNSW existen porque buscar vecinos en espacios vectoriales grandes requiere estructuras especializadas.[47][48]

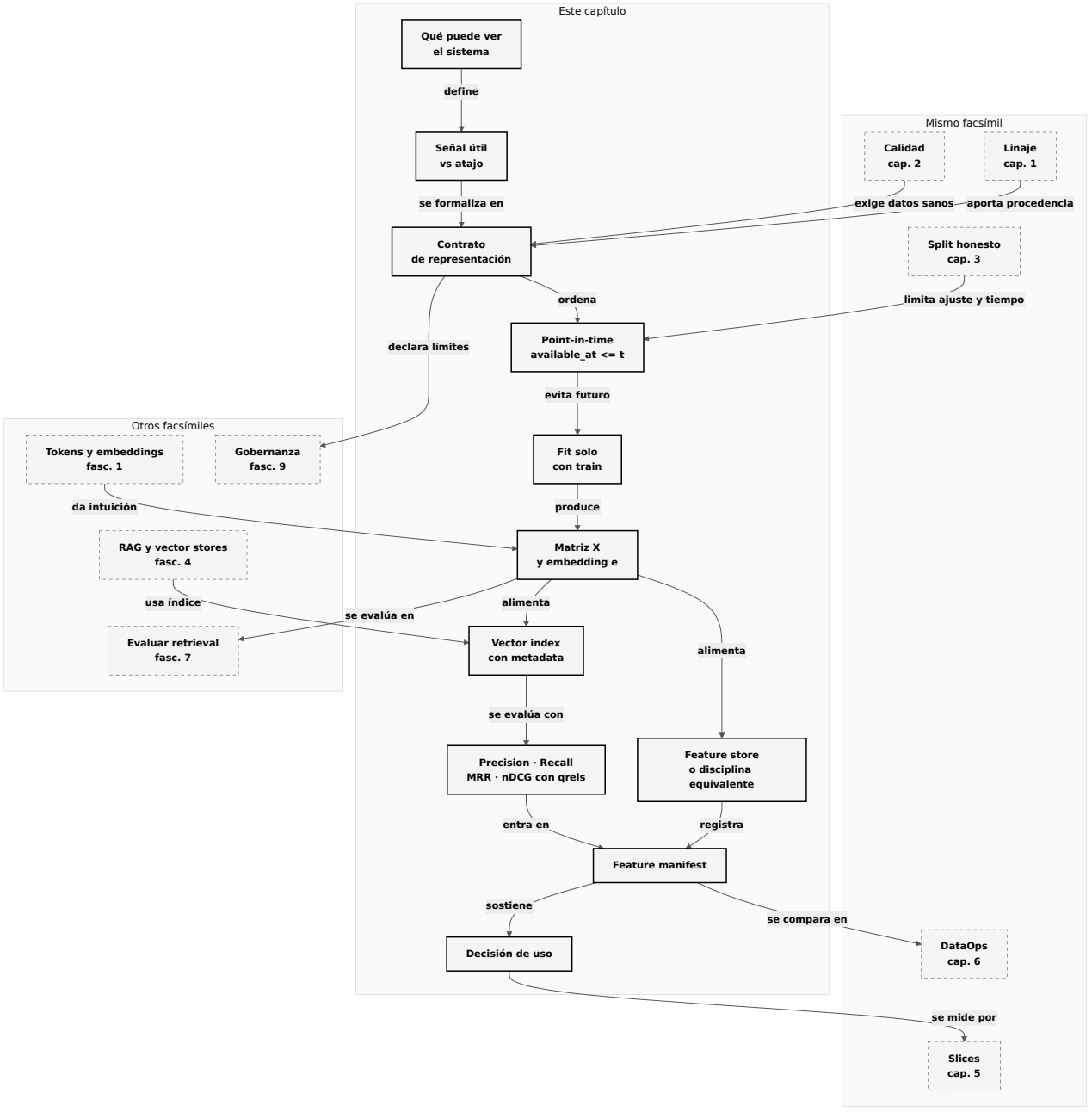
Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Usar todas las columnas	Parece que más información siempre ayuda.	Separar entradas, target, IDs y metadata.
Ajustar vocabulario con todo el dataset	Es más cómodo hacerlo antes del split.	Crear split primero y hacer `fit` solo con train.
Llamar conocimiento a un embedding	El vector parece semántico.	Conservar evidencia, metadata, permisos y evaluación.
Ignorar dimensión y coste	El modelo de embeddings se ve como una caja externa.	Calcular almacenamiento, latencia e índice.
Mirar solo el top-1	El primer vecino puede ser casual.	Evaluar top-k, slices y términos fuera de vocabulario.
Confundir cobertura con recall@k	Una consulta “acierta producto” y parece suficiente.	Crear relevancia anotada antes de comparar retrieval en serio.
Entrenar y servir features distintas	Offline usa SQL histórico y online usa otra API o ventana.	Versionar definiciones y auditar feature skew.

Cómo encaja todo

Este mapa debe leerse como una brújula, no como un inventario de técnicas. Los capítulos anteriores del facsímil nos dieron linaje, calidad y splits; los facsímiles previos nos dieron tokens, embeddings, RAG y vector stores. Este capítulo hace de puente: decide **qué puede ver el sistema** y cómo queda esa decisión convertida en matriz, vector, manifiesto y búsqueda.

La continuidad posterior también es importante. Una representación no termina cuando se genera `feature_matrix.csv`: se evalúa por slices, se monitoriza cuando cambia la producción, se mide como recuperación si entra en RAG y se gobierna si toca permisos, identidad o decisiones sensibles.


Este capítulo
Qué puede ver el sistema
define
Señal útil vs atajo
se formaliza en
Contrato de representación
declara límites
ordena
Point-in-time available_at <= t
evita futuro
Fit solo con train
produce
Matriz X y embedding e
alimenta
alimenta
Vector index con metadata
se evalúa con
Precision · Recall MRR · nDCG con qrels
Feature store o disciplina equivalente
entra en
registra
Feature manifest
sostiene
Decisión de uso
se compara en
se mide por
Mismo facsímil
Calidad cap. 2
Linaje cap. 1
exige datos sanos
aporta procedencia
Split honesto cap. 3
limita ajuste y tiempo
DataOps cap. 6
Slices cap. 5
Otros facsímiles
Tokens y embeddings fasc. 1
Gobernanza fasc. 9
da intuición
RAG y vector stores fasc. 4
usa índice
se evalúa en
Evaluar retrieval fasc. 7

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Feature	Variable de entrada que un modelo puede usar.
Representación	Forma numérica de describir un objeto.
Alta cardinalidad	Situación en la que una variable categórica tiene muchos valores posibles.
One-hot	Codificación binaria de categorías.
Target encoding	Codificación de una categoría usando una estadística del objetivo, siempre con control estricto de leakage.
Normalización	Cambio de escala para comparar valores.
TF-IDF	Peso lexical basado en frecuencia local y rareza global.
Embedding	Vector denso que permite comparar objetos por similitud.
Dimensión	Longitud del vector o número de coordenadas.
Similitud coseno	Comparación de dirección entre dos vectores.
Out-of-vocabulary	Término que no estaba en el vocabulario ajustado.
Leakage de representación	Fuga introducida por transformaciones, selección o estadísticas ajustadas fuera de train.
Feature manifest	Artefacto que versiona columnas, vocabulario, hashes y dimensiones.
Feature skew	Diferencia entre el cálculo de features en entrenamiento y en producción.
Point-in-time correctness	Garantía de que la feature usada estaba disponible cuando se tomó la decisión.
Pipeline	Encadenamiento reproducible de transformaciones y modelo para evitar preprocesado suelto.
Contrato de datos	Reglas ejecutables sobre columnas, tipos, rangos, nulls, categorías y límites de uso.
Vector database	Sistema especializado, o extensión de base de datos, para almacenar y consultar embeddings.
Metadata filter	Filtro estructurado que limita la búsqueda vectorial por permisos, tenant, fecha, producto o fuente.

TÉRMINO	DEFINICIÓN BREVE
Búsqueda híbrida	Combinación de señal lexical, como BM25, y señal vectorial densa.
BM25	Ranking lexical que ajusta frecuencia, rareza global y longitud del documento.
ANN	Búsqueda aproximada de vecinos cercanos para acelerar recuperación vectorial.
HNSW	Índice de búsqueda aproximada basado en grafos jerárquicos.
Product Quantization	Técnica de compresión para búsqueda de vecinos cercanos con menor memoria.
Precision@k	Proporción de resultados relevantes dentro de los k primeros.
Recall@k	Proporción de documentos relevantes recuperados dentro de los k primeros.
MRR	Media del recíproco del ranking del primer resultado relevante.
nDCG@k	Métrica de ranking con relevancia graduada y descuento por posición.
Qrels	Tabla de relevancia anotada para evaluar recuperación de información.
Hard negative	Documento parecido al relevante, pero incorrecto para la consulta.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Por qué una feature no es simplemente una columna?
2. ¿Qué diferencia hay entre feature, metadata y target?
3. ¿Por qué one-hot necesita vocabulario ajustado en train?
4. ¿Por qué una categoría de alta cardinalidad no debería ir a one-hot sin pensarlo?
5. ¿Qué riesgo tiene el target encoding si se calcula con todo el dataset?
6. ¿Qué aprende TF-IDF y por qué puede contaminar si se ajusta con todo el dataset?
7. ¿Qué significa que un embedding viva en $\mathbb{R}^m$?
8. ¿Qué mide la similitud coseno?
9. ¿Por qué la dimensión afecta coste y ranking?
10. ¿Qué guarda `feature_manifest.json`?
11. ¿Por qué `search_report.json` queda en `review`?
12. ¿Qué cambiarías para usar embeddings reales sin perder trazabilidad?

13. ¿Qué diferencia hay entre cobertura top-k por producto y recall@k formal?
14. ¿Cuándo usarías MRR y cuándo nDCG@k?
15. ¿Qué es un hard negative y por qué mejora una evaluación de retrieval?
16. ¿Por qué una feature puede ser correcta matemáticamente y aun así romper point-in-time correctness?
17. ¿Sabrías decir de qué tradición viene cada fórmula del capítulo y qué decisión práctica permite tomar?
18. ¿Qué herramienta usarías para validar contrato de datos y cuál usarías para servir features online?
19. ¿Qué parámetros mirarías en un índice HNSW antes de producción?
20. ¿Cuándo elegirías pgvector y cuándo una base vectorial dedicada?
21. ¿Qué metadata mínima exigirías antes de indexar chunks para RAG?

En resumen

IDEA	QUÉ TE LLEVAS
La representación es parte del sistema.	No se improvisa dentro de un notebook.
El `fit` pertenece a train.	Escalas, vocabularios e IDF se congelan antes de validation y test.
No todas las familias de features se tratan igual.	Numéricas, temporales, categóricas, texto y metadata tienen riesgos distintos.
El leakage puede entrar por la transformación.	Selección, escalado, IDF o target encoding pueden mirar test sin que se note.
Embedding no significa verdad.	Es una geometría útil que necesita metadata y evaluación.
La dimensión tiene coste.	Afecta memoria, latencia, índice y calidad de ranking.
Offline y online deben coincidir.	Si entrenamiento y producción calculan distinto, aparece feature skew.
Retrieval necesita relevancia anotada.	Sin `qrels`, la cobertura top-k es una señal inicial, no una métrica final.
El ranking se mide por más de una métrica.	Precision, recall, MRR y nDCG responden preguntas distintas.
Un índice vectorial es infraestructura.	Métrica, HNSW, filtros, cuantización, reindexado y permisos cambian el sistema.
La herramienta no sustituye el contrato.	Pipeline, feature store o vector DB solo ayudan si sabes qué decisión protegen.
El índice necesita metadata.	Sin permisos, versión, fecha y fuente, el embedding es difícil de auditar.
Una práctica real deja artefactos.	Contrato, matriz, manifiesto, reporte y decisión.

Para saber más

Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137-1155. Paper

Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. *SIGIR 2009*, 758-759. DOI

Databricks. (2026). Feature tables in Unity Catalog. Documentación

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT 2019*, 4171-4186. DOI

Feast. (2026). Feast Documentation. Documentación

Feast. (2026). Point-in-time Joins. Documentación

Google Cloud. (2026). Introduction to feature management. Documentación

Great Expectations. (2026). Expectations Overview. Documentación

Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning* (2.ª ed.). Springer. Libro

Johnson, J., Douze, M. y Jégou, H. (2019). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535-547. DOI

Järvelin, K. y Kekäläinen, J. (2002). Cumulated Gain-Based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4), 422-446. DOI

Jégou, H., Douze, M. y Schmid, C. (2011). Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117-128. DOI

Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI

Malkov, Y. A. y Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. DOI

Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv

Milvus. (2026). Filtered Search. Documentación

OpenAI. (2026). Vector embeddings. Documentación

Pandera. (2026). Pandera Documentation. Documentación

pgvector. (2026). pgvector: Open-source vector similarity search for Postgres. GitHub

Pinecone. (2026). Hybrid search. Documentación

Qdrant. (2026). Indexing. Documentación

Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019*, 3982-3992. DOI

Robertson, S. y Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. DOI

scikit-learn. (2026). Common pitfalls and recommended practices. Documentación

scikit-learn. (2026). Column Transformer with Mixed Types. Documentación

scikit-learn. (2026). MinMaxScaler. Documentación

scikit-learn. (2026). Normalizer. Documentación

scikit-learn. (2026). OneHotEncoder. Documentación

scikit-learn. (2026). Pipeline. Documentación

scikit-learn. (2026). Permutation Feature Importance. Documentación

scikit-learn. (2026). TfidfTransformer. Documentación

Tecton. (2026). Construct Training Data. Documentación

Weaviate. (2026). Hybrid search. Documentación

Notas

1. Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning* (2.ª ed.). Springer. Libro. ↩

2. Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI. ↩

3. scikit-learn. (2026). *MinMaxScaler*. Documentación. Consultado el 22 de junio de 2026. ↩

4. scikit-learn. (2026). *Common pitfalls and recommended practices*. Documentación. Consultado el 22 de junio de 2026. ↩

5. scikit-learn. (2026). *Pipeline*. Documentación. Consultado el 22 de junio de 2026. ↩

6. scikit-learn. (2026). *OneHotEncoder*. Documentación. Consultado el 22 de junio de 2026. ↩

7. scikit-learn. (2026). *TfidfTransformer*. Documentación. Consultado el 22 de junio de 2026. ↩

8. Robertson, S. y Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. DOI. ↩

9. Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI. ↩

10. scikit-learn. (2026). *Permutation Feature Importance*. Documentación. Consultado el 21 de junio de 2026. ↩

11. Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning* (2.ª ed.). Springer. Libro. ↩

12. Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137-1155. Paper. ↩

13. Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019*, 3982-3992. DOI. ↩

14. Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137-1155. Paper. ↩

15. Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv. ↩

16. Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT 2019*, 4171-4186. DOI. ↩

17. Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019*, 3982-3992. DOI. ↩

18. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

19. scikit-learn. (2026). *Normalizer*. Documentación. Consultado el 22 de junio de 2026. ↩

20. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

21. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

22. Järvelin, K. y Kekäläinen, J. (2002). Cumulated Gain-Based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4), 422-446. DOI. ↩

23. scikit-learn. (2026). *Column Transformer with Mixed Types*. Documentación. Consultado el 22 de junio de 2026. ↩

24. Pandera. (2026). *Pandera Documentation*. Documentación. Consultado el 6 de junio de 2026. ↩

25. Great Expectations. (2026). *Expectations Overview*. Documentación. Consultado el 6 de junio de 2026. ↩

26. Feast. (2026). *Feature retrieval*. Documentación. Consultado el 22 de junio de 2026. ↩

27. Tecton. (2026). *Construct Training Data*. Documentación. Consultado el 22 de junio de 2026. ↩

28. Databricks. (2026). *Feature tables in Unity Catalog*. Documentación. Consultado el 22 de junio de 2026. ↩

29. Google Cloud. (2026). *Introduction to feature management*. Documentación. Consultado el 22 de junio de 2026. ↩

30. OpenAI. (2026). *Vector embeddings*. Documentación. Consultado el 22 de junio de 2026. ↩

31. pgvector. (2026). *pgvector: Open-source vector similarity search for Postgres*. GitHub. Consultado el 25 de mayo de 2026. ↩

32. Qdrant. (2026). *Indexing*. Documentación. Consultado el 25 de mayo de 2026. ↩

33. Weaviate. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

34. Milvus. (2026). *Filtered Search*. Documentación. Consultado el 25 de mayo de 2026. ↩

35. Pinecone. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

36. Feast. (2026). *Point-in-time Joins*. Documentación. Consultado el 22 de junio de 2026. ↩

37. Tecton. (2026). *Construct Training Data*. Documentación. Consultado el 22 de junio de 2026. ↩

38. Johnson, J., Douze, M. y Jégou, H. (2019). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535-547. DOI. ↩

39. Malkov, Y. A. y Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. DOI. ↩

40. Qdrant. (2026). *Indexing*. Documentación. Consultado el 25 de mayo de 2026. ↩

41. Milvus. (2026). *Filtered Search*. Documentación. Consultado el 25 de mayo de 2026. ↩

42. Jégou, H., Douze, M. y Schmid, C. (2011). Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117-128. DOI. ↩

43. Pinecone. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

4. Weaviate. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

5. Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. *SIGIR 2009*, 758-759. DOI. ↩

6. Feast. (2026). *Feast Documentation*. Documentación. Consultado el 6 de junio de 2026. ↩

7. Johnson, J., Douze, M. y Jégou, H. (2019). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535-547. DOI. ↩

8. Malkov, Y. A. y Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. DOI. ↩

Capítulo 05: Slices, sesgos y decisión algorítmica

Entrando en el tema

En el capítulo anterior convertimos datos en representaciones. Ahora viene una pregunta menos cómoda: **¿esa representación se comporta igual de bien en todas las partes importantes del problema?**

La respuesta casi nunca sale mirando solo la media global. Una accuracy de 0,86, un recall de 0,78 o un coste medio aceptable pueden esconder que el sistema falla en un producto, un canal, un idioma, una fuente de datos, una necesidad de accesibilidad o una combinación pequeña pero importante. A esa partición útil del comportamiento la llamamos slice.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Definir slices útiles.	No partes por columnas al azar: eliges segmentos conectados con la decisión.
Separar atributo de auditoría y feature.	Entiendes que un campo puede servir para medir aunque no deba entrar al modelo.
Calcular métricas por slice.	Obtienes recall, tasa de revisión, falsos positivos, coste y captura segura por grupo.
Leer disparidades sin convertirlas en eslogan.	Sabes qué métrica se compara, qué tamaño de muestra la sostiene y qué acción permite.
Entender criterios clásicos de equidad algorítmica.	Distingues paridad demográfica, igualdad de oportunidad, odds igualadas y calibración por grupo.
Convertir una auditoría en decisión.	Generas un reporte, un CSV de slices, una ficha y una decisión operativa.

La clave es que un slice no es una curiosidad estadística. Es una forma de preguntar si el sistema trata bien las situaciones donde realmente va a usarse. En un producto educativo, puede importar idioma, canal, necesidad de accesibilidad, tipo de trámite, fuente documental o centro. En otro dominio serán otras variables. Lo importante es que cada slice tenga una razón de existir y una acción posible si falla.

La frase central del capítulo:

> Un sistema no se publica porque la media global suene bien; se publica cuando sabes dónde funciona, dónde no y qué harás con esa diferencia.

La escena: la media global llega demasiado tarde

Imagina un sistema que ayuda a priorizar casos académicos. Cada caso recibe un score. Si el score es alto, se prioriza. Si es bajo, sigue el flujo normal. Si cae en medio, se manda a revisión.

La métrica global parece razonable. El sistema acierta muchos casos, revisa una parte manejable y mantiene buena latencia. En una reunión rápida alguien podría decir: “vamos adelante”.

Pero al partir los resultados por slices aparece otra lectura. Los casos con necesidad de adaptación tienen más casos prioritarios enviados al flujo normal. En inglés se revisa mucho más. En un producto concreto, los casos importantes no quedan capturados. La media global no mentía; simplemente comprimía demasiado.

Este capítulo enseña a no dejar que eso pase.

Qué no es auditar por slices

Auditar por slices no es hacer una tabla enorme con todas las columnas del CSV. Si cada valor distinto se convierte en una fila de auditoría, obtienes ruido. Un slice debe existir porque representa una hipótesis: “este canal cambia la forma de escribir”, “este producto tiene más ambigüedad”, “este idioma tiene menos ejemplos”, “este grupo necesita que no confundamos silencio con baja prioridad”.

Tampoco es declarar que una diferencia numérica demuestra una injusticia completa. Una disparidad es una señal medible, no una sentencia universal. Puede venir de datos escasos, etiquetas inconsistentes, política de negocio, diseño del umbral, cobertura desigual, drift o un error real del modelo. La ingeniería consiste en separar esas causas antes de automatizar más.

Y no es lo mismo usar un atributo para decidir que usarlo para auditar. En muchos sistemas, ciertos campos no deben entrar como feature. Aun así, puede ser imprescindible medir resultados agregados por esos campos para detectar si el sistema está fallando justo donde más importa. El contrato debe decirlo: **campo no usado por el modelo, campo permitido para auditoría agregada.**

Qué sí es un slice

Un slice es un subconjunto definido por una condición. Usaremos notación estándar de teoría de conjuntos para nombrarlo. No es una métrica nueva ni un algoritmo: solo una forma compacta de decir “qué filas entran en este segmento de evaluación”. Si D es el conjunto evaluado y A es un atributo de auditoría, el slice asociado al valor a es:

$$D_a = \{(x_i, y_i, \hat{s}_i) \in D : A_i = a\}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
D	Dataset de evaluación.	36 casos de test del cuaderno.
A	Atributo usado para auditar.	`language`.
a	Valor concreto del atributo.	`en`.
D_a	Slice formado por casos con ese valor.	Casos de test en inglés.
x_i	Entrada o representación del caso.	Texto, producto, canal, metadata.
y_i	Etiqueta real.	`true_priority = 1`.
$\hat{s}_i$	Score producido por el sistema.	`0.84`.

La parte importante es que el slice conserva la unidad de decisión. No evaluamos “idioma” en abstracto. Evaluamos **decisiones sobre casos** dentro de un idioma, producto, canal o combinación de condiciones.

En el cuaderno del facsímil, una política de triaje convierte score en decisión. Los umbrales forman parte de la política del ejercicio. Lo importante es que estén congelados antes de mirar test y que cada salida tenga una consecuencia operativa.

REGLA DEL CUADERNO	DECISIÓN	CONSECUENCIA
`score >= 0.78`	Priorizar.	Caso claro de alta prioridad.
`score < 0.38`	Flujo normal.	Caso claro de baja prioridad.
Resto	Revisar.	Caso intermedio que no debería automatizarse sin mirar.

Esta política no se ajusta en test. Test sirve para medir. Si tocamos umbrales mirando el slice que falla, ya no estamos auditando: estamos usando la evaluación como desarrollo. Eso lo vimos en el capítulo 03 del facsímil 08.

Métricas por slice: lo mínimo que un ingeniero debería mirar

Para cada slice conviene calcular una matriz de confusión. En una decisión binaria clásica, tendríamos TP, FP, FN y TN; esa notación es la base habitual para leer clasificadores, curvas ROC, precisión, recall y tasas de error.[1][2] `scikit-learn`, por ejemplo, documenta la matriz de confusión como el conteo de ejemplos reales frente a ejemplos predichos por clase.[3] En una política con revisión, añadimos una tercera salida: `revisar`. Eso cambia la lectura.

Si un caso prioritario se manda a `normal`, tenemos un fallo operativo fuerte. Si se manda a `revisar`, no se ha automatizado bien, pero tampoco se ha dejado pasar sin mirar. Por eso el cuaderno distingue tres métricas:

MÉTRICA	CÁLCULO DEL CUADERNO	LECTURA
Auto-recall	`tp / positives`	De los casos prioritarios, cuántos se priorizan automáticamente. Es recall aplicado solo a la salida `priorizar`.
Miss rate	`fn / positives`	De los casos prioritarios, cuántos se envían a flujo normal. Es la cara operativa del falso negativo.
Captura segura	`(tp + urgent_review) / positives`	De los prioritarios, cuántos quedan priorizados o revisados. Es un cálculo operativo del cuaderno, no un criterio académico de fairness.

Donde:

SÍMBOLO	SIGNIFICADO	EJEMPLO
P	Número de casos positivos reales en el slice.	6 casos prioritarios.
TP	Positivos reales priorizados.	2 casos.
FN	Positivos reales enviados a flujo normal.	4 casos.
`urgent_review`	Positivos reales mandados a revisión.	1 caso.

En palabras: el recall académico pregunta cuántos positivos reales recupera la salida positiva. El cuaderno conserva esa lectura en `auto_recall`, pero añade `captura segura` porque la política no solo decide `priorizar` o `normal`; también puede abstenerse y mandar a revisión. Esa tercera salida no convierte el problema en "justo" por sí misma. Solo evita que ciertos positivos reales se pierdan sin mirar.

Para no olvidar el coste, el cuaderno calcula una lectura operativa separando falsos negativos, falsos positivos y revisiones. Es una tabla de pesos didácticos que el alumno puede cambiar para ver cómo cambia la decisión.

COMPONENTE	QUÉ PENALIZA	PESO DEL CUADERNO	POR QUÉ IMPORTA
`FN`	Enviar un prioritario a flujo normal.	8,0	Es el error más caro del caso.
`FP`	Priorizar un no prioritario.	3,0	Consume capacidad de atención.
`R`	Mandar a revisión humana.	1,2	Protege, pero no escala gratis.

Un coste no tiene que ser euros. Puede ser minutos de equipo, riesgo operativo, saturación de soporte o coste de oportunidad. Lo importante es hacerlo explícito antes de elegir política.

Este párrafo es más importante que la tabla. En IA aplicada, cada decisión desplaza trabajo a algún sitio: al usuario, al equipo de soporte, al equipo legal, al revisor humano, al sistema de colas o al presupuesto de inferencia. Si solo miras `accuracy`, esos desplazamientos desaparecen. Cuando asignas pesos de coste, aunque sean didácticos, estás declarando qué daño estás intentando evitar.

Criterios clásicos: nombres útiles, no dogmas

La literatura de equidad algorítmica distingue varios criterios. Conviene conocerlos porque aparecen en papers, herramientas y auditorías, pero no deben usarse como recetas automáticas.

La **paridad demográfica** compara tasas de selección entre grupos. Es una noción clásica en la literatura de fairness: mira si la salida positiva aparece con tasas parecidas al condicionar por el atributo de grupo, sin comprobar si las tasas base reales son iguales.[4] En nuestro caso sería comparar qué proporción de casos se prioriza en cada slice:

$$P(\hat{Y} = 1 \mid A = a)$$

Sirve para detectar diferencias de tasa de salida. No sabe si los casos positivos reales estaban distribuidos igual. Si un producto recibe más casos realmente prioritarios que otro, exigir la misma tasa de priorización puede ser una mala idea.

La **igualdad de oportunidad** compara la tasa de verdaderos positivos entre grupos, es decir, si los casos positivos reales reciben la salida positiva con tasas parecidas.[5] En el cuaderno se parece al auto-recall por slice:

$$P(\hat{Y} = 1 \mid Y = 1, A = a)$$

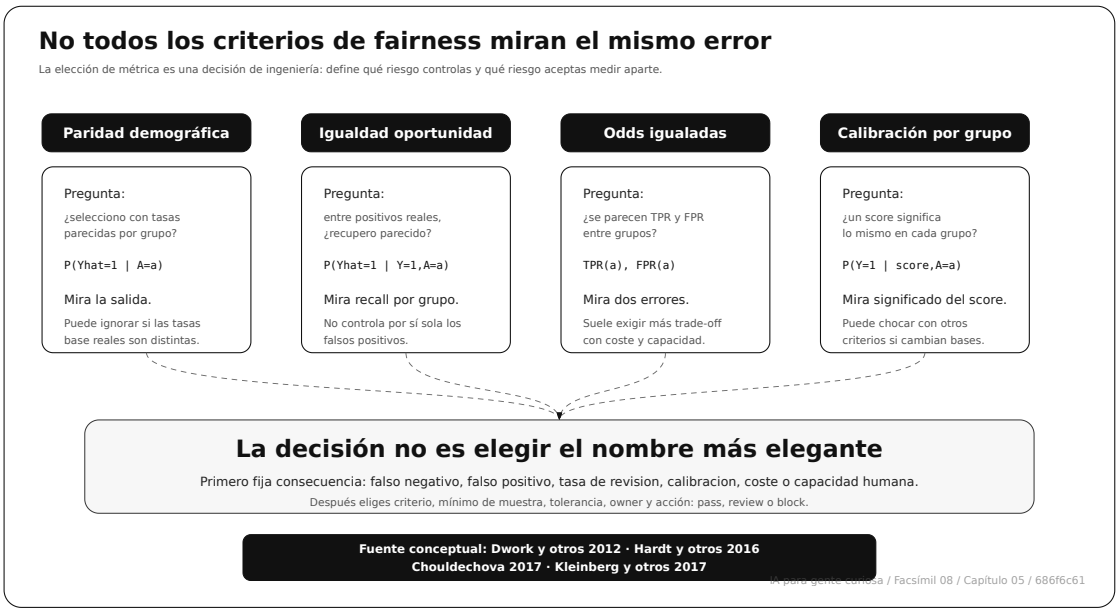
SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{Y}$	Decisión predicha o salida positiva del sistema.	`priorizar`.
Y	Etiqueta real.	`true_priority = 1`.
A	Atributo de grupo o auditoría.	`access_need`.
a	Valor concreto del atributo.	`si`.

En palabras: paridad demográfica mira la tasa de salida positiva dentro de cada grupo; igualdad de oportunidad mira esa tasa solo entre los casos que de verdad eran positivos. La primera responde “¿selecciono igual de a menudo?”. La segunda responde “cuando debía seleccionar, ¿fallo igual de poco?”. Son preguntas distintas y pueden empujar decisiones distintas.

Las **odds igualadas** comparan tanto verdaderos positivos como falsos positivos entre grupos. Pide que el sistema tenga comportamiento parecido para positivos reales y negativos reales. Es más exigente que mirar solo recall.

La **calibración por grupo** pregunta si un score significa lo mismo en cada grupo. Si los casos con score 0,8 aciertan el 80 % en un slice y el 55 % en otro, no basta con decir que el score ordena bien globalmente. Esto conecta con calibración e incertidumbre en el facsímil 07.

Hay un punto clave: algunos criterios no se pueden satisfacer todos a la vez salvo en condiciones especiales. Chouldechova y Kleinberg, Mullainathan y Raghavan mostraron incompatibilidades entre nociones de equidad cuando las tasas base difieren entre grupos.[6][7] Traducido a ingeniería: elegir una métrica de equidad es elegir qué error quieres controlar y qué compromiso aceptas.



Los criterios de equidad algorítmica son preguntas distintas sobre la misma política: salida, positivos reales, falsos positivos, score y coste operativo.

Sesgo no siempre significa lo mismo

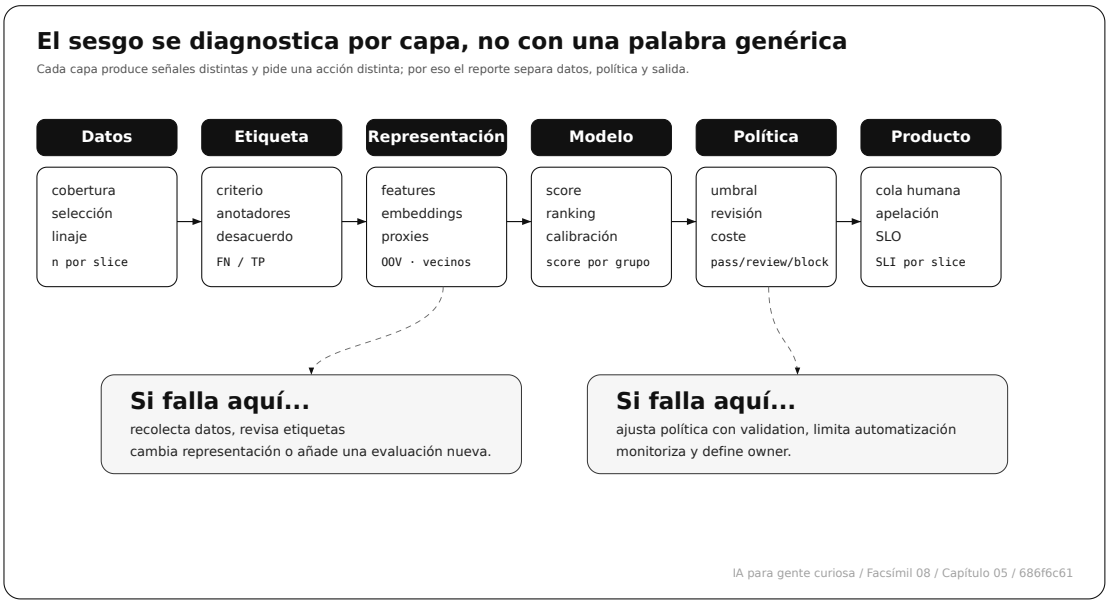
La palabra sesgo se usa demasiado rápido. Para ingeniería, decir “el modelo tiene sesgo” sin precisar de qué tipo es casi no ayuda. Necesitamos localizar **dónde entra**, **cómo se manifiesta**, **qué métrica lo hace visible** y **qué acción permite**.

En este capítulo no usamos sesgo como insulto técnico. Lo usamos como una diferencia sistemática que puede afectar a una decisión. Puede estar en los datos, en el objetivo, en la representación, en el modelo, en el umbral, en la interfaz o en la forma de medir. Si no separas esas capas, acabas arreglando el sitio equivocado.

FUENTE DE SESGO	QUÉ OCURRE	CÓMO SE DETECTA	QUÉ SUELE HACERSE
Cobertura	Un slice tiene pocos ejemplos o no aparece en train.	Conteos, intervalos, unknowns, categorías raras.	Recoger más datos, limitar automatización, declarar cobertura.
Selección	Los datos observados no representan el uso real.	Comparar distribución de train, test y producción.	Rehacer split, muestreo, ponderación o contrato de uso.
Medición	Un campo no mide lo mismo en todos los grupos.	Revisar definición, instrumento de captura y linaje.	Cambiar proxy, añadir variable mejor, documentar límite.
Proxy	Una variable cómoda sustituye mal al concepto real.	Correlación con slices, errores concentrados, revisión de dominio.	Sustituir proxy o cambiar objetivo.
Etiquetado	Las etiquetas se aplican con criterios distintos.	Acuerdo entre anotadores, desacuerdo por slice, revisión de casos frontera.	Reescribir política de anotación y reetiquetar muestra.
Representación	Las features o embeddings capturan peor un segmento.	OOV, vecinos pobres, recall por slice, sensibilidad a idioma o formato.	Cambiar encoder, vocabulario, normalización o datos de entrenamiento.
Agregación	Un único modelo o umbral mezcla subpoblaciones con comportamientos distintos.	Buen promedio global con slices débiles.	Umbrales por contexto, rutas de revisión o modelos especializados con control.
Evaluación	El test no contiene el problema que aparecerá en producción.	Falta de slices críticos, test pequeño, ausencia de intersecciones.	Nueva eval, holdout, monitorización por slice.
Interacción	La interfaz cambia lo que el usuario escribe o aporta.	Diferencias por canal, longitud, idioma, plantilla o formulario.	Rediseñar entrada, instrucciones y validaciones.
Política	El mismo score produce consecuencias distintas según contexto.	Coste, revisión, capacidad operativa y efectos por segmento.	Cambiar umbrales, limitar automatización o exigir revisión.

Un caso clásico de la literatura sanitaria mostró que usar coste sanitario como proxy de necesidad médica podía reducir la identificación de pacientes con necesidades reales en ciertos grupos, porque el gasto histórico no medía necesidad de forma neutral.[8] La lección para este facsímil no es copiar ese dominio; es entender el patrón: **un proxy cómodo puede cambiar la decisión que crees estar midiendo**.

Por eso el capítulo 01 insistía en linaje y uso permitido, el 02 en calidad, el 03 en splits y el 04 en representación. Los slices son donde esas decisiones anteriores se vuelven visibles.



> Un slice ayuda a ubicar la causa probable: no es igual un problema de cobertura que un umbral mal elegido o una cola humana sin capacidad.

Qué sabemos por estudios en modelos reales

Los sesgos no aparecen solo en clasificadores tabulares. También se han observado en embeddings, modelos de lenguaje, clasificadores de texto, sistemas de visión y sistemas de decisión basados en proxies. Ver esos estudios ayuda porque cambia la pregunta: ya no es “¿puede pasar?”, sino “¿cómo lo detectaría en mi caso?”.

ESTUDIO	TIPO DE MODELO O SISTEMA	QUÉ MOSTRÓ	SEÑAL DETECTABLE
Bolukbasi y otros (2016)	Word embeddings	Algunas analogías en embeddings capturaban asociaciones de género no deseadas.[9]	Direcciones en el espacio vectorial y vecinos semánticos.
Caliskan y otros (2017)	Embeddings entrenados con corpus	WEAT mostró asociaciones recuperables desde lenguaje ordinario.[10]	Test de asociación entre conjuntos de términos.
Buolamwini y Gebru (2018)	Clasificación facial comercial	Las tasas de error variaban mucho por intersección de tono de piel y género en sistemas evaluados.[11]	Accuracy por grupos e intersecciones, no solo global.
Dixon y otros (2018)	Clasificación de toxicidad	Ciertos términos de identidad podían disparar predicciones tóxicas aunque el texto no lo fuera.[12]	Falsos positivos asociados a términos concretos.
CrowS-Pairs (2020)	Modelos de lenguaje enmascarados	Comparó pares de frases para medir preferencias por frases con estereotipo.[13]	Probabilidad relativa entre pares mínimos.
StereoSet (2021)	Modelos de lenguaje preentrenados	Midió sesgo estereotípico junto con capacidad de modelado lingüístico.[14]	Relación entre score lingüístico y score de estereotipo.
BBQ (2022)	Pregunta-respuesta	Evaluó si modelos recurren a estereotipos cuando el contexto es insuficiente o ambiguo.[15]	Respuestas bajo contexto ambiguo frente a contexto informativo.

Hay dos lecciones de ingeniería en esa tabla.

La primera: el sesgo no siempre aparece como “métrica baja”. A veces aparece como asociación geométrica, falso positivo lexical, peor cobertura en intersecciones, sensibilidad al contexto ambiguo o score calibrado de forma distinta por grupo.

La segunda: cada tipo de sistema necesita su prueba. Para embeddings haces vecinos, direcciones, WEAT o pares mínimos. Para clasificación haces matriz por slice. Para RAG miras recuperación por fuente, idioma, fecha y permisos. Para generación miras pares contrafactuales, respuestas con contexto insuficiente, abstención y criterios de evaluación.

Detectabilidad: cómo se ve un sesgo antes de producción

Una auditoría útil no pregunta “¿hay sesgo?” en abstracto. Pregunta “¿qué señal observable esperaría si este sistema falla de forma sistemática?”.

SEÑAL DETECTABLE	CÓMO SE CALCULA	QUÉ INDICA	QUÉ NO DEMUESTRA SOLA
Slice con poco n	Conteo por segmento.	Falta evidencia para concluir.	Que el sistema sea malo en ese slice.
Diferencia de recall	`tp / positives` por slice.	Un grupo de positivos reales se detecta peor.	La causa del problema.
Diferencia de falsos positivos	`fp / negatives` por slice.	Un grupo recibe más salidas positivas indebidas.	Que el objetivo esté bien definido.
Captura segura baja	`(tp + urgent_review) / positives`.	Casos importantes pasan sin priorizar ni revisar.	Qué componente lo provocó.
Tasa de revisión dispar	`review / n` por slice.	Un segmento se automatiza menos o consume más revisión.	Que revisar sea necesariamente malo.
Coste por caso dispar	`cost_total / n` por slice.	El impacto operativo se concentra.	Que el coste esté bien ponderado.
OOV alto	Términos fuera del vocabulario por slice.	La representación cubre peor un segmento.	Que un embedding real no lo arregle.
Vecinos pobres	Top-k sin evidencia útil para un slice.	Recuperación débil o índice mal cubierto.	Que el generador sea el único culpable.
Pares contrafactuales inestables	Cambiar solo un atributo y comparar salida.	Sensibilidad indeseada a un cambio controlado.	Que todos los casos reales fallen igual.
Contexto ambiguo decide demasiado	Probar preguntas con evidencia insuficiente.	El modelo rellena huecos con asociaciones aprendidas.	Que falle cuando la evidencia es completa.

La palabra “detectable” importa. Si un sesgo no tiene métrica, muestra, caso de prueba o traza, se convierte en debate interminable. El objetivo no es tener todas las respuestas, sino diseñar señales que permitan revisar.

Por eso conviene escribir la prueba antes de mirar resultados. Si el equipo decide los slices después de ver la tabla, puede acabar contando la historia que más le conviene. Si los decide antes, con una hipótesis clara, la auditoría se parece más a ingeniería y menos a búsqueda de

relato. La pregunta no es “qué diferencia puedo encontrar”, sino “qué diferencia sería peligrosa si existiera”.

Buenas prácticas para no fabricar un problema nuevo

La buena práctica empieza antes de entrenar. Un equipo serio no espera al final para “pasar fairness”. Diseña la evaluación desde el contrato de datos.

BUENA PRÁCTICA	QUÉ HACES EN CONCRETO
Definir la unidad de decisión	No mezclas usuario, caso, documento, chunk y sesión como si fueran lo mismo.
Separar feature y auditoría	Un campo puede no entrar al modelo y aun así medirse de forma agregada.
Escribir slices críticos antes de test	Evitas elegir segmentos solo porque cuentan una historia cómoda.
Medir intersecciones	`language=en` puede parecer aceptable y `language=en + access_need=si` no.
Añadir mínimos de muestra	Sin n, positivos y negativos suficientes, el slice va a revisión.
Guardar política y hashes	Si cambia el dataset o el contrato, cambia la auditoría.
Congelar umbrales antes de test	Ajustar con test convierte auditoría en desarrollo.
Mirar coste, no solo métrica	Un fallo raro puede importar más que diez aciertos baratos.
Dejar una acción escrita	`pass`, `review` o `block` con razones y siguiente paso.
Conectar con monitorización	Los mismos slices se observan después en producción.

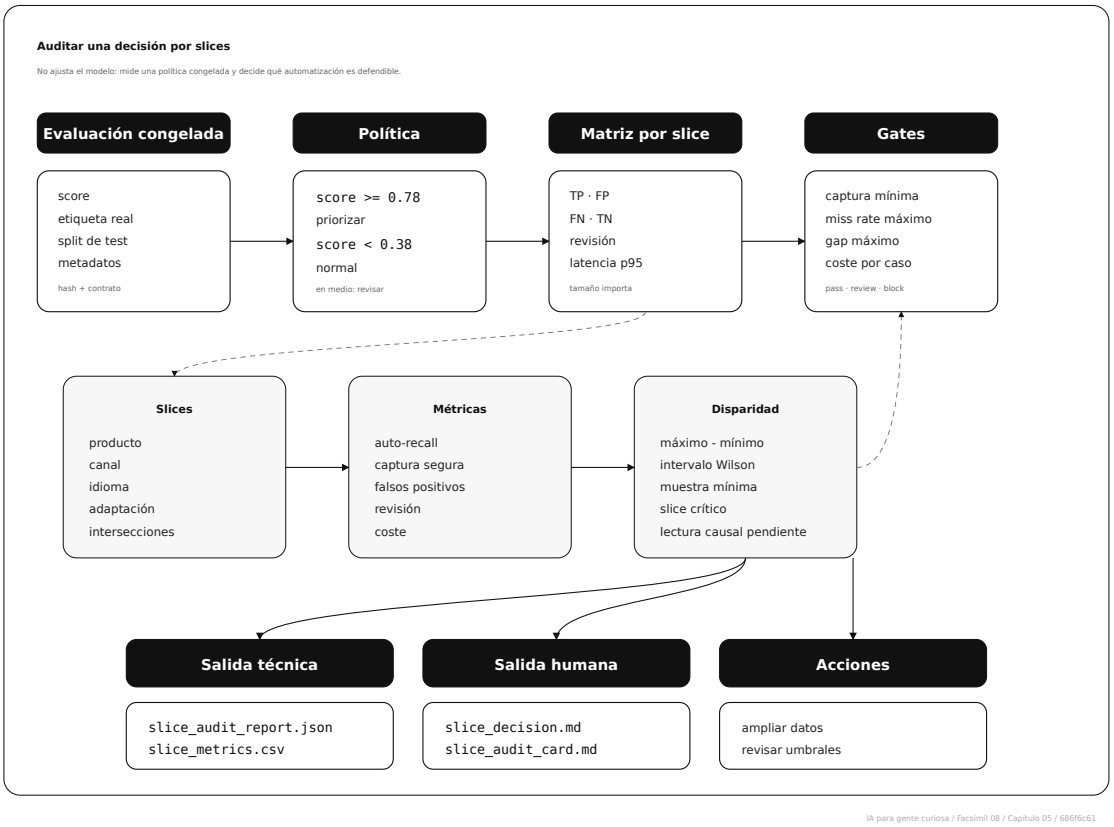
Y hay cosas que conviene evitar:

EVITA	POR QUÉ
“No medimos ese atributo, así que no hay problema”	No usar un campo como feature no implica que el sistema no tenga diferencias por ese campo.
“La muestra es pequeña, pero la media global pasa”	Un slice pequeño no se arregla escondiéndolo dentro del promedio.
“El benchmark público dice que el modelo es bueno”	El benchmark quizá no cubre tu dominio, idioma, interfaz o coste.
“Mitigamos borrando una columna”	Otros campos pueden actuar como proxies.
“Probamos muchos slices y publicamos solo los interesantes”	Eso convierte auditoría en selección de relato.
“Una herramienta lo certifica”	La herramienta calcula; el equipo define uso, consecuencia, coste, límites y acción.
“Arreglamos el test tocando el umbral”	Si el umbral se elige mirando test, necesitas nueva evaluación.
“Todos los slices deben tener la misma tasa”	Algunas tasas base reales pueden diferir; la pregunta es qué error estás controlando.

La buena práctica completa siempre termina en una decisión concreta. Si un slice queda débil por falta de muestra, quizá no publicas, pero tampoco concluyes que el sistema sea injusto: pides más datos o dejas el segmento en revisión. Si un slice falla con muestra suficiente, toca elegir: cambiar representación, revisar etiquetas, ajustar política en validation, añadir revisión humana o limitar automatización. Sin esa acción escrita, la auditoría se queda en una tabla incómoda que nadie sabe convertir en trabajo.

El cuaderno del facsímil incorpora estas prácticas en pequeño: `fields_not_for_model`, `audit_fields`, `critical_slices`, mínimos de muestra, gates, hashes y decisión Markdown. No es una auditoría completa de un sistema real, pero sí tiene la forma profesional que debe tener una primera revisión.

Anatomía de una auditoría de decisión



Auditar por slices convierte una métrica global en una decisión revisable: datos, política, segmentos, métricas, gates y acciones.

Herramientas reales y cómo leerlas

Fecha de corte de esta lectura: **7 de junio de 2026**. El mercado cambia rápido, así que no conviene memorizar logos. Conviene memorizar **qué artefacto produce cada herramienta**: métrica por slice, intervalo, explicación, alerta, comparación de versiones, política de mitigación o evidencia para una revisión.

Una herramienta seria no “quita el sesgo” en abstracto. Hace una de estas cosas: mide una diferencia, ayuda a encontrar causa probable, aplica una técnica de mitigación, deja trazabilidad o vigila si producción se aleja de lo que mediste en validación. Si no produce un artefacto revisable, es una demo bonita, no una práctica de ingeniería.

Herramientas abiertas para auditar y mitigar

Estas herramientas encajan bien cuando el equipo trabaja con notebooks, `pandas`, `scikit-learn`, TensorFlow o reportes reproducibles en CI. Son especialmente útiles para enseñar y para construir un primer estándar interno, porque puedes inspeccionar datos, métricas y código.

HERRAMIENTA	DÓNDE ENCAJA	QUÉ PERMITE VER	QUÉ PUEDE AYUDAR A MITIGAR	QUÉ NO DEBES DELEGARLE
Fairlearn	Modelos tabulares o pipelines compatibles con `scikit-learn`.	Métricas agrupadas con `MetricFrame`, comparación entre grupos y visualización de disparidades.[16]	Reducciones como `ExponentiatedGradient`, búsqueda con restricciones y optimización de umbrales.[17]	La definición de grupos, coste del error y métrica que gobierna la decisión.
AI Fairness 360	Comparar algoritmos de fairness, datasets de referencia y métricas clásicas.	Métricas, detectores, explicadores y API compatible con `scikit-learn` para parte del toolkit.[18]	Preprocesamiento, técnicas durante entrenamiento y postprocesamiento.[19]	El encaje con tu contrato de datos real. Muchos ejemplos académicos no se parecen a tu producto.
Aequitas	Auditoría de scores binarios o continuos con CSV, CLI o interfaz local.	Métricas de grupo, disparidades y criterios sobre scores, etiquetas y atributos.[20]	No mitiga por sí sola: ayuda a decidir si el modelo o la política deben cambiar.	Convertir una tabla de disparidades en decisión sin revisar el dominio.
Fairness Indicators	Ecosistema TensorFlow, TFMA y evaluación por slices.	Visualización de rendimiento por segmentos, intervalos y métricas en pipelines TensorFlow.[21]	No es una varita de mitigación; empuja a localizar slices débiles y a rediseñar datos o entrenamiento.	Pensar que solo aplica a “temas sociales”. También sirve para idioma, canal, dispositivo o fuente.
What-If Tool	Exploración interactiva de ejemplos, umbrales y contrafactuales.	Cómo cambian predicciones al mover ejemplos, atributos o umbrales.[22]	Ayuda a descubrir hipótesis antes de automatizar una prueba.	Sustituir una evaluación versionada. Explorar no es certificar.
Responsibly	Aprendizaje, prototipos y análisis de fairness en clasificación y NLP.	Métricas, visualizaciones y utilidades para enseñar conceptos con código.[23]	Puede acompañar un primer laboratorio o auditoría exploratoria.	Producción sin controles propios, versionado y trazabilidad.

Para un curso de ingeniería, Fairlearn y AIF360 son buenas primeras estaciones porque obligan a escribir `y_true`, `y_pred`, score, atributo sensible o de auditoría y métrica. Ese gesto parece básico, pero es la mitad del aprendizaje: si no sabes pasar esos arrays con nombres correctos, todavía no sabes qué estás auditando.

Plataformas cloud y de ciclo de vida

Cuando el sistema ya vive en una nube concreta, las herramientas integradas aportan algo que una libreta local no tiene: conexión con jobs, registros de modelo, monitorización, permisos, reportes y gobierno de releases.

PLATAFORMA	QUÉ CUBRE	CUÁNDO TIENE SENTIDO	CUIDADO DE INGENIERÍA
Amazon SageMaker Clarify	Sesgo en datos y modelos, explicabilidad, evaluaciones de modelos fundacionales y monitorización integrada con SageMaker.[24]	Si tu entrenamiento, registro o despliegue ya está en SageMaker.	Hay que fijar qué features o atributos se analizan, en qué split, con qué baseline y qué acción dispara una alerta.
Azure Responsible AI dashboard	Paneles de análisis responsable dentro de Azure ML: errores, importancia de variables, contrafactuales, causalidad y fairness según el tipo de modelo.[25]	Si trabajas con Azure ML y quieres que evaluación, registro y explicación vivan en el mismo flujo.	Un panel no sustituye un gate en CI/CD. Debe acabar en una decisión reproducible.
SageMaker Model Monitor, Azure ML monitoring o equivalentes	Vigilancia de datos, predicciones, latencia y cambios por segmento.	Si el riesgo no termina al desplegar.	La métrica por slice debe existir antes de producción; no la inventes cuando salta la alerta.

La pregunta correcta para una nube no es “¿tiene dashboard?”. Es: **¿puedo versionar la política, reproducir el reporte, bloquear un release, abrir una incidencia y comparar producción contra validación por los mismos slices?** Si no, el panel sirve para mirar, pero no para gobernar.

Observabilidad y herramientas de mercado

En producción aparece otra familia: plataformas de observabilidad de ML y GenAI. No suelen ser la primera herramienta para aprender fairness, pero sí son relevantes cuando tienes tráfico real, múltiples modelos, trazas, embeddings, RAG, usuarios, feedback y alertas.

HERRAMIENTA	QUÉ SUELE APORTAR	DÓNDE AYUDA CON SESGOS	QUÉ PEDIR ANTES DE ADOPTARLA
Evidently	Librería y plataforma para calidad de datos, drift, evaluación y reportes.[26]	Comparar referencia frente a producción, crear reportes por segmento y vigilar si cambian distribuciones.	Exportar reportes, integrarlo en CI y definir tus propios slices, no solo métricas por defecto.
Giskard	Evaluación y testing de modelos, agentes y aplicaciones GenAI.[27]	Pruebas de comportamiento en LLMs: toxicidad, estereotipos, robustez de prompts, RAG y regresiones de comportamiento.	Datasets de prueba propios y criterios explícitos. Sin casos de tu dominio, solo pruebas una maqueta.
Fiddler	Observabilidad, explicabilidad y métricas de fairness en producción.[28]	Monitorizar rendimiento por cohortes, explicar predicciones y detectar degradación por grupos.	Acceso a etiquetas o feedback posterior; sin ground truth retrasado, algunas señales serán aproximadas.
Arize	Observabilidad para ML, visión, embeddings y GenAI, con slicing, drift y análisis de rendimiento.[29]	Slices, cohortes, embeddings, drift y degradación de rendimiento en producción.	Que los atributos de auditoría lleguen al sistema con permisos y granularidad correcta.
WhyLabs	Observabilidad con perfiles de datos, drift, calidad, modelos predictivos y GenAI.[30]	Vigilar cambios de datos, problemas de calidad, drift, outputs de LLM y trazas.	Que las alertas estén conectadas con owners, SLOs y acciones, no solo con correos.
Arthur, WhyLabs, Fiddler, Arize u otros proveedores enterprise	Gobierno, monitorización, dashboards, explicabilidad y control operativo.[31]	Cuando hay varios equipos, varios modelos y requisitos de trazabilidad.	Evita comprar “confianza”. Compra integración, auditoría, permisos, exportación y comparabilidad.

El mercado es útil cuando resuelve un problema de operación: logging, trazas, control de versiones, dashboards compartidos, alertas, permisos, exportaciones y soporte. Pero el mercado no conoce por defecto tu coste de error. Tampoco sabe si un falso positivo molesta, si un falso negativo deja a alguien sin atención o si un slice con poca muestra debe bloquear o pedir más datos.

Mitigar no es borrar una columna

Hay una trampa clásica: detectar una diferencia, borrar el atributo sensible y declarar el problema resuelto. En sistemas reales, los proxies aparecen por código postal, idioma, canal, horario, dispositivo, historial, coste, texto libre o fuente documental. Por eso la mitigación

debe elegir **dónde está la causa probable**.

CAPA DE MITIGACIÓN	QUÉ CAMBIA	EJEMPLOS TÉCNICOS	CUÁNDO USARLA	QUÉ VIGILAR
Datos	La distribución, cobertura o calidad de las muestras.	Recolectar más casos, reetiquetar, balancear, reponderar, mejorar instrucciones de anotación, separar fuentes.	Hay slices con poca muestra, etiquetas inconsistentes o proxies claros.	No fabricar un dataset artificial que ya no se parezca a producción.
Representación	Cómo se codifica el caso.	Mejorar features, revisar embeddings, normalizar idioma, añadir metadatos permitidos, cambiar encoder, reducir OOV.	El modelo no ve señales suficientes en ciertos grupos o formatos.	No meter campos que expliquen demasiado bien una condición sensible sin justificación.
Entrenamiento	La función objetivo o restricciones del modelo.	Fairlearn reductions, restricciones de paridad, regularización, pérdidas ponderadas, búsqueda de hiperparámetros con gates.	El modelo aprende una frontera útil pero desigual.	Medir trade-off: una restricción puede mejorar un slice y empeorar otro.
Umbral y política	La conversión de score a acción.	Umbral global, banda de revisión, umbrales condicionados con revisión de dominio, abstención, doble lectura humana.	El score está razonablemente calibrado, pero la acción automática concentra riesgo.	Cambiar umbrales por grupo tiene implicaciones de producto, ética, regulación y soporte. No se improvisa.
Producto	La experiencia y el circuito humano.	Explicación al usuario, apelación, revisión manual, colas por capacidad, feedback posterior, monitorización.	La consecuencia de equivocarse es alta o no hay datos suficientes para automatizar.	Que la revisión humana no sea un cajón sin dueño, SLA ni trazabilidad.

La mitigación buena suele ser aburrida: cambiar datos, contratos, umbrales, monitorización, revisión y documentación. La mala mitigación suele parecer elegante: un algoritmo nuevo aplicado sin explicar qué consecuencia reduce.

Cómo elegir herramienta según tu stack

SI ESTÁS AQUÍ	EMPIEZA POR	AÑADE DESPUÉS
Notebook con CSV y modelo `sklearn`	Fairlearn o Aequitas para métricas por grupo.	AIF360 si necesitas comparar técnicas de mitigación.
Pipeline TensorFlow/TFX	Fairness Indicators y What-If Tool.	Reporte versionado y gate por slice en CI.
SageMaker	SageMaker Clarify.	Model Monitor, Model Cards y job recurrente por slice.
Azure ML	Responsible AI dashboard.	Componente de evaluación que exporte métricas a tu release.
Producto con tráfico real	Evidently, Fiddler, Arize, WhyLabs o plataforma equivalente.	SLO por slice, owner, alerta, runbook y comparación contra validación.
LLM, agente o RAG	Giskard, Evidently, Arize Phoenix u otras evals propias.	Dataset de pares mínimos, prompts de regresión, trazas y revisión humana en casos críticos.
Auditoría académica o de clase	Cuaderno propio + Fairlearn/AIF360/Aequitas.	Informe reproducible, model card, data card y defensa oral de decisiones.

Para decidir, yo haría esta prueba: toma diez filas reales, un contrato de política y una métrica por slice. Si la herramienta no puede decirte **qué fila falló, en qué slice, con qué métrica, contra qué baseline, desde qué versión y qué acción toca**, aún no tienes una herramienta de auditoría; tienes una visualización.

Qué evitar al comprar o adoptar una herramienta

1. Elegir por capturas de pantalla. Hay que pedir exportaciones, API, reproducibilidad y ejemplos con tus datos.
2. Medir fairness solo en validación y olvidarlo en producción. La distribución cambia y los slices también.
3. Usar atributos de auditoría sin política de acceso. Medir requiere permiso, minimización y propósito.
4. Mitigar antes de diagnosticar. No es igual falta de cobertura, proxy, mala etiqueta, mala calibración o umbral mal puesto.
5. Comparar herramientas sin el mismo dataset, split, métrica y política de decisión.
6. Aceptar métricas por defecto sin traducirlas a coste. Paridad, recall, falsos positivos y calibración no responden la misma pregunta.

7. Confundir un panel verde con una decisión publicable. La salida debe ser `pass`, `review` o `block`, con evidencia.

La regla práctica es sencilla: usa herramientas, pero no les delegues el criterio. Una herramienta calcula, visualiza, alerta o ejecuta una técnica. El equipo decide qué métrica representa la consecuencia que quiere controlar, qué slices importan y qué se hace cuando algo no pasa.

Caso completo: de `block` a `review`

Un buen capítulo de sesgos no puede quedarse en "usa una herramienta". Tiene que enseñar una historia completa: una política parece razonable, falla en slices críticos, se propone una mitigación y se vuelve a medir. Eso es lo que hace el cuaderno.

La política base dice:

DECISIÓN	REGLA
`priorizar`	`score >= 0.78`
`normal`	`score < 0.38`
`revisar`	`0.38 <= score < 0.78`

Con esa política, el resultado global queda en `block`: captura segura `0.7778`, pérdida operativa `0.2222`, tasa de revisión `0.3611` y coste por caso `1.4056`. El problema no es solo la media. En los slices críticos aparece algo peor:

SLICE CRÍTICO	N	POSITIVOS	AUTO-RECALL	PÉRDIDA	CAPTURA SEGURA	REVISIÓN	COSTE POR CASO
`language=en`	9	4	0.25	0.25	0.75	0.5556	1.5556
`access_need=si`	12	6	0.0	0.6667	0.3333	0.5	3.2667
`product=practicas	access_need=si`	4	2	0.0	1.0	0.0	0.5

La lectura humana es directa: algunos casos prioritarios con necesidad de accesibilidad o en inglés están yendo a flujo normal. No basta con decir "la accuracy global es aceptable". La política está dejando pasar casos que el contrato considera críticos.

La mitigación candidata no cambia el modelo. Cambia la política de decisión: amplía la banda de revisión bajando el umbral de flujo normal de `0.38` a `0.32`.

DECISIÓN	POLÍTICA BASE	POLÍTICA CANDIDATA
`priorizar`	`score >= 0.78`	`score >= 0.78`
`normal`	`score < 0.38`	`score < 0.32`
`revisar`	`0.38 <= score < 0.78`	`0.32 <= score < 0.78`

El antes/después queda así:

POLÍTICA	ESTADO	CAPTURA SEGURA	PÉRDIDA OPERATIVA	TASA DE REVISIÓN	COSTE POR CASO	FLAGS BLOCK	FLAGS REVIEW
Base	`block`	0.7778	0.2222	0.3611	1.4056	5	7
Banda de revisión	`review`	1.0	0.0	0.5278	0.7167	0	5

Esto no significa “ya está arreglado”. Significa algo más interesante para ingeniería: la candidata elimina los casos prioritarios enviados a flujo normal, pero aumenta la carga de revisión humana. Pasa de `block` a `review`, no a `pass`. Si el equipo no puede revisar aproximadamente el 53 % de los casos, la mitigación mejora una métrica y rompe la operación. Por eso fairness, coste y capacidad deben leerse juntos.

La matemática mínima de la mitigación

En un proyecto serio no se elige mitigación porque “suena mejor”. La literatura de clasificación justa con restricciones, por ejemplo el enfoque de reducciones de Agarwal y otros, formula el problema como minimizar error bajo restricciones de equidad medibles.[32] Una forma académica de leerlo es:

$$\min_{h \in \mathcal{H}} \quad \operatorname{error}(h) \text{ s.t.} \qquad \gamma_j(h) \leq \epsilon_j, \quad j = 1, \ldots, J$$

Donde:

SÍMBOLO	QUÉ SIGNIFICA	EJEMPLO
h	Clasificador o política candidata.	Política base o banda de revisión.
$\mathcal{H}$	Familia de clasificadores considerados.	Políticas que cambian umbrales o modelos candidatos.
$\text{error}(h)$	Error que se quiere minimizar.	Falsos negativos, falsos positivos o coste definido.
s.t.	Abreviatura estándar de *subject to*.	Introduce las restricciones que la solución no puede romper.
$\gamma_j(h)$	Violación de la restricción j, por ejemplo una disparidad de oportunidad.	Gap de recall por encima de lo permitido.
ϵ_j	Tolerancia permitida para esa restricción.	Gap máximo aceptado por el gate.

En palabras: la optimización no busca solo “el modelo que menos se equivoca en promedio”. Busca una política que reduzca error sin violar restricciones medibles. Si una política candidata tiene $\gamma_j(h) = 0{,}18$ para una restricción cuyo límite es $\epsilon_j = 0{,}10$, la violación no es una opinión: supera el margen permitido en `0.08`. En un sistema real eso debería activar `review` o `block`, según el contrato.

La parte importante es la restricción. No queremos solo minimizar error global. Queremos minimizarlo sin romper restricciones de slices, coste y operación. Fairlearn lo expresa con reducciones y restricciones; AIF360 ofrece varias familias de mitigación; nuestro cuaderno lo enseña con algo más humilde pero más transparente: umbrales, gates, slices y coste.

Hay tres familias de respuesta:

FAMILIA	QUÉ OPTIMIZA	EJEMPLO EN EL CUADERNO	CUÁNDO SIRVE
Cambiar datos	Mejorar cobertura o etiquetas.	Recolectar más casos `access_need=si`.	Cuando el slice falla porque casi no hay evidencia fiable.
Cambiar modelo	Modificar entrenamiento o representación.	Reentrenar con features, embeddings o pérdidas mejor controladas.	Cuando el score separa mal en ciertos segmentos.
Cambiar política	Modificar umbrales, revisión o automatización.	Ampliar banda de revisión de `0.38` a `0.32`.	Cuando el score es incierto y la consecuencia de mandar a flujo normal es alta.

La mitigación por política suele ser la primera que un equipo puede probar porque no exige reentrenar. También es peligrosa si se usa sin capacidad: cada caso enviado a revisión debe tener owner, SLA, registro y criterio de cierre.

De validación a producción

La auditoría no termina al generar `slice_metrics.csv`. En producción pueden cambiar los canales, idiomas, productos, prompts, documentos, colas humanas o criterios de etiqueta. Por eso los mismos slices deben convertirse en señales operativas.

SEÑAL EN PRODUCCIÓN	QUÉ MIDE	QUÉ ACCIÓN DEBERÍA DISPARAR
`review_rate` por slice	Carga humana que genera la política.	Ajustar capacidad, revisar umbral o limitar automatización.
`miss_rate` con etiqueta retrasada	Casos importantes que acabaron en flujo normal.	Abrir revisión de política y bloquear aumento de automatización.
Drift de distribución por slice	Cambio en canales, idiomas, productos o perfiles.	Comparar contra validación y revisar datos de entrenamiento.
Latencia p95 por slice	Si ciertos casos tardan más en resolverse.	Revisar pipeline, herramientas o colas específicas.
Tasa de apelación o corrección	Si usuarios o revisores corrigen más un segmento.	Revisar etiqueta, interfaz, rúbrica o explicación.

Esta es la conexión natural con DataOps: un SLI por slice no es una frase bonita, es una medida que se calcula. Un SLO por slice no es “queremos hacerlo bien”, es un objetivo que decide si se mantiene, se revisa o se detiene una política.

En el día a día

En un proyecto de IA aplicada, los slices deben definirse antes de mirar el resultado final. Puedes añadir nuevos slices cuando aparece una señal nueva, pero no deberías probar veinte cortes hasta encontrar el que cuenta la historia que querías contar.

Una plantilla mínima de decisión sería:

PREGUNTA	RESPUESTA QUE DEBE EXISTIR
¿Cuál es la unidad de decisión?	Caso, usuario, documento, consulta, turno, sesión.
¿Qué campos se usan como features?	Lista permitida y lista prohibida.
¿Qué campos se usan solo para auditoría?	Segmentos agregados, con permiso y propósito claro.
¿Qué slices son críticos?	Los que no pueden fallar aunque la media global pase.
¿Qué métrica gobierna cada slice?	Recall, miss rate, revisión, coste, latencia, calibración.
¿Qué gate decide?	Umbral medible de pass, review o block.
¿Qué acción sigue si falla?	Más datos, revisión de etiqueta, cambio de umbral, desautomatización, monitorización.

Para un ingeniero, lo más peligroso no es que el reporte diga `block`. Lo peligroso es que no exista reporte. Sin slices, la decisión parece técnica porque tiene números, pero no tiene trazabilidad.

Por qué debería importarte

Una auditoría por slices no es una capa estética de “IA responsable”. Es una herramienta de ingeniería para no publicar una política que falla justo donde el coste del error es más alto. Si el sistema prioriza bien en promedio, pero deja pasar casos críticos en un segmento pequeño, la media global te está contando una verdad incompleta.

También importa por operación. Cada mitigación cambia alguna carga: más revisión humana, más datos, más latencia, más coste de etiquetado, menos automatización o más complejidad de gobierno. Por eso este capítulo no trata solo de detectar diferencias: trata de convertir esas diferencias en decisiones defendibles. A veces la decisión será publicar. A veces será publicar con monitorización. A veces será ampliar datos. Y a veces será bloquear.

La pregunta profesional no es “¿hay sesgo?”. La pregunta es: **qué slice falla, con qué métrica, con qué muestra, con qué coste, qué acción dispara y quién se hace cargo**. Si no puedes contestar eso, todavía no tienes una auditoría; tienes una tabla.

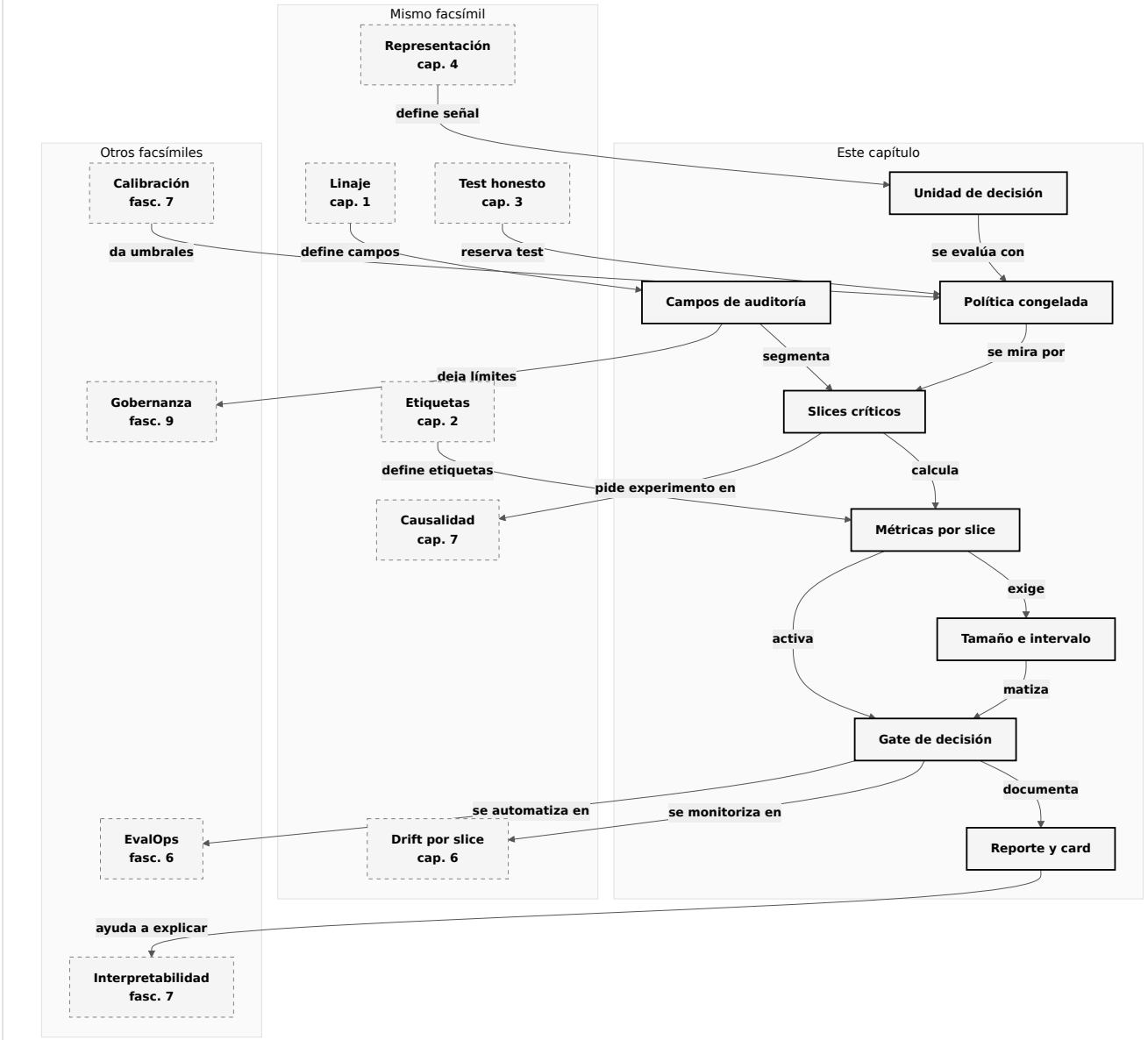
Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Mirar solo la media global	Es cómoda y fácil de explicar.	Exigir slices críticos antes de aprobar.
Partir por demasiadas columnas	Parece más completo.	Empezar por hipótesis de decisión y coste.
Confundir auditoría con feature	Si tengo el campo, parece que puedo usarlo.	Separar `fields_not_for_model` y `audit_fields`.
Ajustar umbrales mirando test	Es tentador arreglar el slice que falla.	Congelar umbrales desde validation y versionar la política.
Ignorar tamaños pequeños	Una tabla con números da sensación de rigor.	Añadir mínimos, intervalos y estado `review`.
Elegir una métrica de equidad sin contexto	La palabra suena suficiente.	Preguntar qué error controla y qué compromiso acepta.
No dejar acción concreta	El reporte se queda en diagnóstico.	Escribir decisión, dueño, gate y siguiente paso.

Cómo encaja todo

Este mapa debe leerse como continuidad de ingeniería. Los capítulos anteriores construyeron el suelo: linaje, calidad, split honesto y representación. Este capítulo pregunta si la política resultante se comporta de forma defendible en los segmentos que importan. Después, el capítulo 06 usará estos mismos slices como señales de monitorización: si producción cambia, no miraremos solo drift global.

La decisión central aquí es pasar de “mi métrica global mejora” a “sé qué slices sostienen o impiden automatizar”. Esa decisión conecta directamente con evaluación, calibración, interpretabilidad y gobernanza.


Mismo facsímil
Representación
cap. 4
define señal
Otros facsímiles
Calibración
fasc. 7
Linaje
cap. 1
Test honesto
cap. 3
Este capítulo
Unidad de decisión
da umbrales
define campos
reserva test
se evalúa con
Campos de auditoría
Política congelada
segmenta
se mira por
deja límites
Gobernanza
fasc. 9
Etiquetas
cap. 2
Slices críticos
define etiquetas
pide experimento en
calcula
Causalidad
cap. 7
Métricas por slice
exige
activa
Tamaño e intervalo
matiza
Gate de decisión
documenta
se automatiza en
se monitoriza en
EvalOps
fasc. 6
Drift por slice
cap. 6
Reporte y card
ayuda a explicar
Interpretabilidad
fasc. 7

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Slice	Subconjunto de datos definido por una condición útil para evaluar una decisión.
Atributo de auditoría	Campo usado para medir comportamiento por segmentos.
Atributo no usado por el modelo	Campo que no entra como feature, pero puede usarse para auditoría agregada.
Paridad demográfica	Comparación de tasas de selección entre grupos.
Igualdad de oportunidad	Comparación de verdaderos positivos entre grupos positivos reales.
Odds igualadas	Comparación simultánea de verdaderos positivos y falsos positivos entre grupos.
Calibración por grupo	Comprobación de que un score significa algo parecido en cada grupo.
Disparidad	Diferencia medible entre slices para una métrica concreta.
Captura segura	Proporción de casos importantes priorizados o enviados a revisión.
Gate	Regla medible que convierte métricas en decisión de release.
Proxy	Variable que sustituye a otra más difícil de medir, con riesgo de medir otra cosa.
OOV	Término fuera del vocabulario usado para construir una representación textual.
Par mínimo	Dos entradas casi iguales que solo cambian una pieza controlada.
WEAT	Test de asociación para medir relaciones entre grupos de palabras en embeddings.
Intersección	Slice definido por la combinación de varios atributos.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Qué es un slice y por qué no equivale a cualquier columna?

2. ¿Qué diferencia hay entre feature y atributo de auditoría?
3. ¿Por qué una métrica global puede esconder un problema importante?
4. ¿Qué mide la paridad demográfica?
5. ¿Qué mide la igualdad de oportunidad?
6. ¿Por qué odds igualadas mira verdaderos positivos y falsos positivos?
7. ¿Qué significa que algunas nociones de equidad sean incompatibles en ciertos escenarios?
8. ¿Por qué conviene separar auto-recall y captura segura?
9. ¿Qué coste asignarías a un falso negativo operativo y por qué?
10. ¿Por qué `access_need=si` debería bloquear una política de triaje?
11. ¿Qué archivo contiene la tabla plana de métricas por slice?
12. ¿Qué harías si un slice crítico tiene muestra insuficiente?
13. ¿Por qué no se deben ajustar umbrales mirando test?
14. ¿Cómo conectarías esta auditoría con una model card?
15. ¿Qué slices monitorizarías en producción en el capítulo siguiente?
16. ¿Qué diferencia hay entre sesgo de cobertura, medición, proxy y representación?
17. ¿Por qué los estudios de embeddings no se detectan igual que los de clasificación?
18. ¿Qué señal usarías para detectar falsos positivos asociados a términos concretos?
19. ¿Qué significa probar pares mínimos o contrafactuales?
20. ¿Qué parte del playbook adaptarías primero a tu proyecto?
21. ¿Por qué la política candidata pasa de `block` a `review`, pero no a `pass`?
22. ¿Qué trade-off aparece al ampliar la banda de revisión?
23. ¿Qué representa $\gamma_j(h)$ en una mitigación con restricciones?
24. ¿Qué pedirías a una herramienta antes de incorporarla al ciclo de release?
25. ¿Qué SLI por slice monitorizarías cuando el sistema llegue a producción?

En resumen

IDEA	QUÉ TE LLEVAS
La media global no decide sola.	Hay que mirar slices conectados con consecuencias reales.
Un campo puede auditar sin decidir.	No todo atributo útil para medir debe entrar al modelo.
La métrica depende del error.	Paridad, oportunidad, odds y calibración responden preguntas distintas.
El tamaño importa.	Un slice pequeño pide revisión o más datos, no conclusión fuerte.
La auditoría debe producir acción.	Reporte, CSV, card y decisión son artefactos de ingeniería.
Los sesgos tienen mecanismos distintos.	Cobertura, proxy, etiqueta, representación y política requieren pruebas diferentes.
Las buenas prácticas se escriben antes de test.	Si eliges slices y gates después, la auditoría pierde fuerza.
Mitigar implica trade-offs.	La banda de revisión reduce pérdida, pero aumenta carga humana.
Las herramientas no deciden solas.	Fairlearn, AIF360, Clarify o Evidently ayudan si hay contrato, slices y gates.
Producción cambia la pregunta.	Los mismos slices deben convertirse en SLI, SLO, alerta y runbook.

Para saber más

Agarwal, A., Beygelzimer, A., Dudík, M., Langford, J. y Wallach, H. (2018). A Reductions Approach to Fair Classification. *ICML*, 60-69. PMLR

Amazon Web Services. (2026). *Amazon SageMaker Clarify*. Documentación

Arize AI. (2026). *ML Observability Platform*. Producto

Arthur AI. (2020). *Product Update - Bias Monitoring v2.1*. Blog

Bellamy, R. K. E., Dey, K., Hind, M., Hoffman, S. C., Houde, S., Kannan, K., Lohia, P., Martino, J., Mehta, S., Mojsilovic, A., Nagar, S., Ramamurthy, K. N., Richards, J., Saha, D., Sattigeri, P., Singh, M., Varshney, K. R. y Zhang, Y. (2019). AI Fairness 360: An Extensible Toolkit for Detecting and Mitigating Algorithmic Bias. *IBM Journal of Research and Development*, 63(4/5), 4:1-4:15. DOI

Bolukbasi, T., Chang, K.-W., Zou, J. Y., Saligrama, V. y Kalai, A. T. (2016). Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings. *Advances in Neural Information Processing Systems 29*, 4349-4357. Paper

Buolamwini, J. y Gebru, T. (2018). Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification. *Proceedings of Machine Learning Research*, 81, 77-91. Paper

Caliskan, A., Bryson, J. J. y Narayanan, A. (2017). Semantics Derived Automatically from Language Corpora Contain Human-Like Biases. *Science*, 356(6334), 183-186. DOI

Center for Data Science and Public Policy. (2026). *Aequitas documentation*. Documentación

Chouldechova, A. (2017). Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data*, 5(2), 153-163. DOI

Dixon, L., Li, J., Sorensen, J., Thain, N. y Vasserman, L. (2018). Measuring and Mitigating Unintended Bias in Text Classification. *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, 67-73. Google Research

Dwork, C., Hardt, M., Pitassi, T., Reingold, O. y Zemel, R. (2012). Fairness Through Awareness. *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 214-226. DOI

Evidently AI. (2026). *Evidently documentation*. Documentación

Fairlearn. (2026). *Assessment: Performing a Fairness Assessment*. Documentación

Fairlearn. (2026). *Mitigations*. Documentación

Fawcett, T. (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8), 861-874. DOI

Fiddler AI. (2026). *Fairness*. Documentación

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. DOI

Giskard. (2026). *Giskard documentation*. Documentación

Hardt, M., Price, E. y Srebro, N. (2016). Equality of Opportunity in Supervised Learning. *Advances in Neural Information Processing Systems 29*, 3323-3331. Paper

IBM Research. (2026). *AI Fairness 360 documentation*. Documentación

Kleinberg, J., Mullainathan, S. y Raghavan, M. (2017). *Inherent Trade-Offs in the Fair Determination of Risk Scores*. arXiv

Microsoft. (2026). *Use the Responsible AI dashboard in Azure Machine Learning studio*. Documentación

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. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220-229. DOI

Nadeem, M., Bethke, A. y Reddy, S. (2021). StereoSet: Measuring Stereotypical Bias in Pretrained Language Models. *ACL-IJCNLP 2021*, 5356-5371. DOI

Nangia, N., Vania, C., Bhalerao, R. y Bowman, S. R. (2020). CrowS-Pairs: A Challenge Dataset for Measuring Social Biases in Masked Language Models. *EMNLP 2020*, 1953-1967. DOI

Obermeyer, Z., Powers, B., Vogeli, C. y Mullainathan, S. (2019). Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations. *Science*, 366(6464), 447-453. DOI

Parrish, A., Chen, A., Nangia, N., Padmakumar, V., Phang, J., Thompson, J., Htut, P. M. y Bowman, S. R. (2022). BBQ: A Hand-Built Bias Benchmark for Question Answering. *Findings of ACL 2022*, 2086-2105. DOI

Powers, D. M. W. (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies*, 2(1), 37-63. arXiv

Raji, I. D., Smart, A., White, R. N., Mitchell, M., Gebru, T., Hutchinson, B., Smith-Loud, J., Theron, D. y Barnes, P. (2020). *Closing the AI Accountability Gap: Defining an End-to-End Framework for Internal Algorithmic Auditing*. arXiv

Responsibly. (2026). *Responsibly documentation*. Documentación

Scikit-learn. (2026). *Classification metrics*. Documentación

Scikit-learn. (2026). *confusion_matrix*. Documentación

TensorFlow. (2026). *Fairness Indicators*. GitHub

Wexler, J., Pushkarna, M., Bolukbasi, T., Wattenberg, M., Viégas, F. y Wilson, J. (2019). The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE Transactions on Visualization and Computer Graphics*. Google Research

WhyLabs. (2026). *WhyLabs documentation*. Documentación

Notas

1. Fawcett, T. (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8), 861-874. https://doi.org/10.1016/j.patrec.2005.10.010 ↩

2. Powers, D. M. W. (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies*, 2(1), 37-63. https://arxiv.org/abs/2010.16061 ↩

3. Scikit-learn. (2026). *confusion_matrix*. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html. Consultado el 7 de junio de 2026. ↩

4. Dwork, C., Hardt, M., Pitassi, T., Reingold, O. y Zemel, R. (2012). Fairness Through Awareness. *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 214-226. https://doi.org/10.1145/2090236.2090255 ↩

5. Hardt, M., Price, E. y Srebro, N. (2016). Equality of Opportunity in Supervised Learning. *Advances in Neural Information Processing Systems 29*, 3323-3331. https://papers.nips.cc/paper/6374-equality-of-opportunity-in-supervised-learning ↩

6. Chouldechova, A. (2017). Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data*, 5(2), 153-163. https://doi.org/10.1089/big.2016.0047 ↩

7. Kleinberg, J., Mullainathan, S. y Raghavan, M. (2017). *Inherent Trade-Offs in the Fair Determination of Risk Scores*. https://arxiv.org/abs/1609.05807 ↩

8. Obermeyer, Z., Powers, B., Vogeli, C. y Mullainathan, S. (2019). Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations. *Science*, 366(6464), 447-453. https://doi.org/10.1126/science.aax2342 ↩

9. Bolukbasi, T., Chang, K.-W., Zou, J. Y., Saligrama, V. y Kalai, A. T. (2016). Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings. *NeurIPS 2016*, 4349-4357. https://papers.nips.cc/paper/6228-man-is-to-computer-programmer-as-woman-is-to-homemaker-debiasing-word-embeddings ↩

10. Caliskan, A., Bryson, J. J. y Narayanan, A. (2017). Semantics Derived Automatically from Language Corpora Contain Human-Like Biases. *Science*, 356(6334), 183-186. https://doi.org/10.1126/science.aal4230 ↩

11. Buolamwini, J. y Gebru, T. (2018). Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification. *Proceedings of Machine Learning Research*, 81, 77-91. https://proceedings.mlr.press/v81/buolamwini18a.html ↩

12. Dixon, L., Li, J., Sorensen, J., Thain, N. y Vasserman, L. (2018). Measuring and Mitigating Unintended Bias in Text Classification. *AIES 2018*, 67-73. https://research.google/pubs/measuring-and-mitigating-unintended-bias-in-text-classification/ ↩

13. Nangia, N., Vania, C., Bhalerao, R. y Bowman, S. R. (2020). CrowS-Pairs: A Challenge Dataset for Measuring Social Biases in Masked Language Models. *EMNLP 2020*, 1953-1967. https://doi.org/10.18653/v1/2020.emnlp-main.154 ↩

.4. Nadeem, M., Bethke, A. y Reddy, S. (2021). StereoSet: Measuring Stereotypical Bias in Pretrained Language Models. *ACL-IJCNLP 2021*, 5356-5371. https://doi.org/10.18653/v1/2021.acl-long.416 ↩

.5. Parrish, A. y otros (2022). BBQ: A Hand-Built Bias Benchmark for Question Answering. *Findings of ACL 2022*, 2086-2105. https://doi.org/10.18653/v1/2022.findings-acl.165 ↩

.6. Fairlearn. (2026). *Assessment: Performing a Fairness Assessment*. https://fairlearn.org/main/user_guide/assessment/. Consultado el 7 de junio de 2026. ↩

.7. Fairlearn. (2026). *Mitigations*. https://fairlearn.org/main/user_guide/mitigation/index.html. Consultado el 7 de junio de 2026. ↩

.8. IBM Research. (2026). *AI Fairness 360 documentation*. https://aif360.readthedocs.io/en/stable/index.html. Consultado el 7 de junio de 2026. ↩

.9. Bellamy, R. K. E. y otros (2019). AI Fairness 360: An Extensible Toolkit for Detecting and Mitigating Algorithmic Bias. *IBM Journal of Research and Development*, 63(4/5), 4:1-4:15. https://doi.org/10.1147/JRD.2019.2942287 ↩

0. Center for Data Science and Public Policy. (2026). *Aequitas documentation*. https://dssg.github.io/aequitas/. Consultado el 7 de junio de 2026. ↩

1. TensorFlow. (2026). *Fairness Indicators*. https://github.com/tensorflow/fairness-indicators. Consultado el 7 de junio de 2026. ↩

2. Wexler, J. y otros (2019). The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE TVCG*. https://research.google/pubs/the-what-if-tool-interactive-probing-of-machine-learning-models/ ↩

3. Responsibly. (2026). *Responsibly documentation*. https://docs.responsibly.ai/. Consultado el 7 de junio de 2026. ↩

4. Amazon Web Services. (2026). *Amazon SageMaker Clarify*. https://aws.amazon.com/sagemaker/ai/clarify/. Consultado el 7 de junio de 2026. ↩

5. Microsoft. (2026). *Use the Responsible AI dashboard in Azure Machine Learning studio*. https://learn.microsoft.com/en-us/azure/machine-learning/how-to-responsible-ai-dashboard. Consultado el 7 de junio de 2026. ↩

6. Evidently AI. (2026). *Evidently documentation*. https://docs.evidentlyai.com/docs/library/overview. Consultado el 7 de junio de 2026. ↩

7. Giskard. (2026). *Giskard documentation*. https://docs.giskard.ai/. Consultado el 7 de junio de 2026. ↩

8. Fiddler AI. (2026). *Fairness*. https://docs.fiddler.ai/observability/fairness. Consultado el 7 de junio de 2026. ↩

29. Arize AI. (2026). *ML Observability Platform*. https://arize.com/capabilities/. Consultado el 7 de junio de 2026. ↩

30. WhyLabs. (2026). *WhyLabs documentation*. https://docs.whylabs.ai/docs/. Consultado el 7 de junio de 2026. ↩

31. Arthur AI. (2020). *Product Update - Bias Monitoring v2.1*. https://www.arthur.ai/blog/product-update-bias-monitoring-v21. Consultado el 7 de junio de 2026. ↩

32. Agarwal, A., Beygelzimer, A., Dudík, M., Langford, J. y Wallach, H. (2018). A Reductions Approach to Fair Classification. *ICML*, 60-69. https://proceedings.mlr.press/v80/agarwal18a.html ↩

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.

Slices y sesgo: el promedio miente

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2008/05-slices-sesgo-el-promedio-miente.ipynb

Capítulo 06: DataOps: pipelines, drift y monitorización

Entrando en el tema

En el capítulo anterior vimos que una decisión no se publica solo porque tenga una métrica global aceptable. Hay que mirarla por slices, escribir gates y decidir `pass`, `review` o `block`. Ahora viene la pregunta operativa: **¿qué pasa después de publicar?**

Los datos cambian. Las colas cambian. Los idiomas cambian. Un producto nuevo concentra tráfico. Un campo deja de llegar. Una traza se pierde. Una latencia sube. La etiqueta real llega días después. Y el sistema que el lunes parecía razonable puede dejar de representar la producción del jueves.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar un pipeline de datos para IA.	Distingues fuente, validación, transformación, evaluación, serving y monitorización.
Leer una ejecución como artefacto.	Sabes qué inputs, outputs, hashes, versiones y estado debe dejar una run.
Definir SLIs y SLOs de datos.	Escribes indicadores medibles, objetivos y acciones cuando fallan.
Medir drift sin convertirlo en superstición.	Calculas distancia entre referencia y producción y decides qué revisar.
Conectar slices con operación.	No miras solo drift global: mides slices críticos y consecuencias.
Crear un runbook y un postmortem.	Dejas una guía de respuesta y una explicación técnica de la incidencia.

La promesa de este capítulo es operativa: cuando algo cambie, no deberías empezar desde cero. Deberías poder abrir una run, ver qué datos entraron, qué contrato los validó, qué versión del pipeline los transformó, qué métricas se degradaron, qué slice concentra el daño y qué runbook toca ejecutar. Eso no elimina incidentes; elimina improvisación.

La frase central:

DataOps no es tener dashboards. Es saber qué dato cambió, qué run lo produjo, qué decisión afecta y qué acción toca.

La escena: ayer pasaba, hoy bloquea

Imagina que el asistente académico ya está en producción. El viernes se aprobó la política de decisión: casos claros se priorizan, casos claramente normales pasan a flujo normal y casos inciertos van a revisión. El lunes todo parece estable.

El martes entra una campaña de prácticas internacionales. De repente suben los casos en inglés, casi todos con necesidad de accesibilidad, y además una parte del pipeline pierde `trace_id`. La métrica global del mes todavía no parece dramática, pero la ventana del día cuenta otra historia: más latencia, más revisión, más casos prioritarios que acaban como `normal`.

Ese es el problema que resuelve DataOps. No pregunta solo “¿funcionó el modelo?”. Pregunta:

PREGUNTA OPERATIVA	POR QUÉ IMPORTA
¿Qué ventana cambió?	Una media mensual puede esconder un día roto.
¿Qué versión de pipeline produjo los eventos?	Un cambio de código puede explicar el problema.
¿Qué datos entraron?	Sin hash ni versión, no se reproduce la decisión.
¿Qué slices fallan?	El daño operativo suele concentrarse.
¿Hay trazas completas?	Sin `trace_id`, investigar se vuelve conjetura.
¿Qué runbook se ejecuta?	Una alerta sin acción se convierte en ruido.

Estas preguntas enseñan una idea sencilla: un incidente de datos rara vez tiene una sola cara. Puede ser cambio de distribución, fallo de instrumentación, retraso de etiquetas, rotura de contrato, aumento de latencia o mezcla de todo lo anterior. Si el sistema solo tiene una alerta global, cada incidente se convierte en discusión. Si tiene trazas, versiones y SLIs, la conversación empieza mucho más cerca de la causa.

Qué no es DataOps

DataOps no es “tener un CSV limpio”. Tampoco es instalar un orquestador y dar por resuelta la operación. Un orquestador lanza jobs; no decide por sí mismo si un dataset representa producción, si un slice crítico se degradó o si una ventana debe bloquear un release.

Tampoco es guardar logs infinitos sin contrato. Si cada evento tiene mil campos pero no sabemos cuáles son obligatorios, qué SLI alimentan o quién responde cuando fallan, hemos cambiado desorden pequeño por desorden caro.

Y no es mirar drift como si fuera una alarma universal. Drift significa cambio de distribución. A veces es una incidencia; a veces es una campaña esperada; a veces es una mejora de cobertura; a veces indica que la evaluación ya no representa el uso real. La métrica abre una revisión. No sustituye el criterio.

Qué sí es DataOps para IA

DataOps es tratar los datos, runs y artefactos como piezas operables del sistema. Esto incluye contratos, validaciones, linaje, versionado, monitorización, gates y runbooks. En IA, además, debe conectar con modelos, features, embeddings, prompts, evals, slices y decisiones.

Sculley y otros explicaron que los sistemas de ML acumulan deuda técnica por dependencias de datos, cambios no locales y realimentaciones difíciles de ver.[1] TFX formalizó una arquitectura de producción con ingestión, validación, transformación, entrenamiento, evaluación y serving reproducibles.[2] El ML Test Score propuso que la madurez de un sistema ML se mida también por tests de datos, monitorización y gestión de cambios.[3]

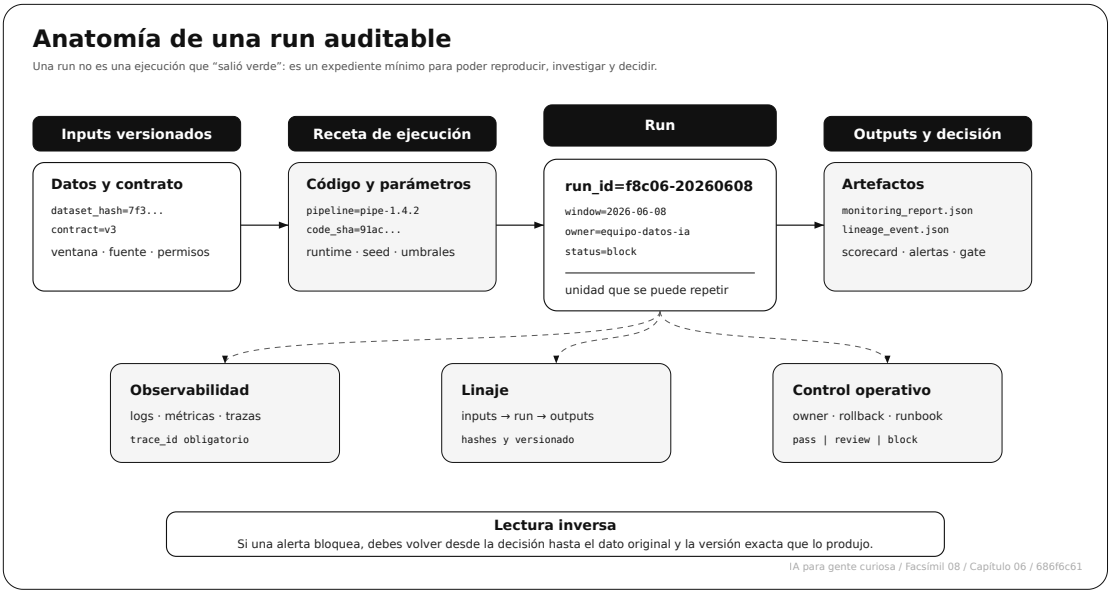
Fecha de corte: **7 de junio de 2026**. Fuentes consultadas ese día: OpenLineage, OpenTelemetry, Great Expectations, Evidently, Prometheus, Apache Airflow, Dagster, Prefect, dbt, TFX, ML Test Score y literatura de ingeniería de ML. Lo estable no es la herramienta concreta: es la obligación de versionar, medir, trazar y decidir.

El pipeline por dentro

Un pipeline de IA no es solo “entrenar y desplegar”. Es una cadena de contratos. Cada organización tendrá pasos distintos; lo importante no es adornarlo con símbolos, sino que cada paso transforme una entrada, produzca una salida y deje evidencia auditable.

PASO	ENTRADA	SALIDA	EVIDENCIA MÍNIMA
Ingesta	Fuente externa o interna.	Snapshot controlado.	`source_id`, timestamp, hash.
Validación	Snapshot.	Reporte de calidad.	Checks, columnas, valores inválidos.
Transformación	Datos válidos.	Features, chunks o embeddings.	Versión de código, parámetros, hashes.
Evaluación	Predicciones o outputs.	Métricas y slices.	Dataset, política, umbrales, intervalos.
Gate	Métricas.	`pass`, `review` o `block`.	Regla exacta y motivo.
Serving	Modelo o política.	Decisiones.	`trace_id`, versión, latencia.
Monitorización	Eventos de producción.	Alertas y scorecard.	SLI, SLO, runbook, owner.

La tabla se lee de izquierda a derecha, pero también al revés. Si una alerta dice que `2026-06-08` queda en `block`, deberías poder volver desde la decisión hasta la run, desde la run hasta los inputs, desde los inputs hasta los hashes y desde los hashes hasta el contrato que dice por qué ese dato era aceptable o no. Esa reversibilidad es la diferencia entre “tengo un dashboard” y “puedo defender técnicamente lo que ha pasado”.



> Una run auditable conecta entradas, contrato, versión de código, parámetros, outputs, trazas, linaje y decisión.

OpenLineage propone un estándar abierto para capturar metadatos de runs, jobs y datasets en componentes de pipeline.[4] OpenTelemetry define señales de observabilidad como trazas, métricas y logs; una traza permite seguir una operación por spans, una métrica mide valores agregados y un log registra eventos.[5][6][7]

En términos prácticos:

SEÑAL	QUÉ CONTESTA	EJEMPLO EN IA
Log	Qué ocurrió.	"Evento p013 llegó sin `trace_id`".
Métrica	Cuánto ocurre.	`missing_trace_rate = 0.1`.
Traza	Por dónde pasó.	Ingesta → validación → decisión → revisión.
Linaje	De dónde viene y qué produjo.	`production_events.csv` → `monitoring_report.json`.
Runbook	Qué hacer ahora.	Revisar pipeline `pipe-1.4.2`, idioma `en`, slice `access_need=si`.

Patrones de arquitectura de pipeline

Un ingeniero de software no debería quedarse en “tengo un script”. La pregunta es qué garantías necesita ese script cuando se ejecuta todos los días, con datos que cambian y con personas dependiendo de sus salidas.

PATRÓN	QUÉ RESUELVE	RIESGO SI NO LO DISEÑAS
Batch	Procesar ventanas cerradas.	Repetir una ventana y obtener otra decisión sin saber por qué.
Streaming	Procesar eventos conforme llegan.	Mezclar eventos tardíos, duplicados o incompletos.
Checkpoint	Recordar hasta dónde llegó una run.	Reprocesar a ciegas o perder eventos.
Backfill	Recalcular ventanas antiguas.	Cambiar histórico sin versionar pipeline, datos y contrato.
Replay	Reproducir eventos con una versión concreta.	No poder depurar una decisión pasada.
Retry	Reintentar un paso fallido.	Duplicar efectos si no hay idempotencia.
Canary	Probar una versión en una parte controlada.	Publicar un cambio a toda producción sin señal temprana.
Shadow mode	Ejecutar una versión sin afectar la decisión.	Confundir evaluación silenciosa con decisión real.

Apache Airflow organiza workflows como DAGs con tareas y dependencias.[8] Dagster enfatiza assets definidos por software, linaje, observabilidad y testabilidad.[9] Prefect documenta deployments con metadatos de ejecución remota, versiones y configuración, y también retries configurables por workflow o tarea.[10][11]

La herramienta importa menos que el contrato de ejecución:

DECISIÓN DE DISEÑO	PREGUNTA QUE DEBES RESPONDER
Orquestador	¿Quién lanza la run, con qué parámetros y dónde queda su estado?
Idempotencia	¿Qué clave evita duplicar una decisión si se reintenta?
Retries	¿Qué pasos se pueden repetir y cuáles requieren revisión manual?
Backfill	¿Qué versión de datos, código y contrato se usa para recalcular?
Checkpoint	¿Qué pasa si una run cae a mitad?
Output	¿La salida se escribe una vez, se sobreescribe o se versiona?
Rollback	¿Cómo volvemos a una política anterior sin perder trazabilidad?

Idempotencia merece una explicación propia. Una operación es idempotente si repetirla produce el mismo efecto observable que ejecutarla una vez. En pipelines de IA, la clave suele ser algo como:

```
idempotency_key = window + event_id
```

Si una tarea falla después de emitir una decisión y se reintenta sin esa clave, puedes duplicar un caso, mandar dos revisiones humanas o contar dos veces una alerta. Esto no es un detalle de plataforma: es ingeniería de software aplicada al dato.

Herramientas por capa, sin casarse con ninguna

En DataOps conviene hablar de herramientas porque el alumno las va a encontrar en empresas, prácticas, ofertas de trabajo y proyectos reales. Pero la herramienta no debe aparecer como tótem. Airflow, Dagster o Prefect no arreglan un contrato mal definido; Great Expectations no sabe qué coste tiene un falso negativo; OpenTelemetry no inventa el `trace_id` que tu aplicación no emitió; Prometheus no decide si una ventana debe bloquear un release. La herramienta hace visible y repetible una disciplina que ya deberías haber diseñado.

CAPA	HERRAMIENTAS HABITUALES	QUÉ APORTAN	PREGUNTA DE INGENIERÍA ANTES DE USARLAS
Orquestación	Airflow, Dagster, Prefect	Ejecutar dependencias, reintentos, scheduling y estado de runs.	¿Qué pasa si una tarea falla después de escribir salida parcial?
Contratos y calidad	dbt contracts, Great Expectations	Declarar columnas, tipos, checks y expectativas.	¿Qué cambio de schema rompe consumidores y cuál es compatible?
Linaje	OpenLineage e integraciones de orquestadores	Relacionar jobs, runs, datasets, inputs y outputs.	¿Puedo reconstruir qué produjo una decisión concreta?
Observabilidad	OpenTelemetry, Prometheus, Grafana	Emitir trazas, métricas, logs y alertas.	¿Qué SLI dispara acción y quién responde?
Drift y monitorización ML	Evidently, dashboards de ML observability	Comparar referencia y producción, slices y cambios de distribución.	¿El drift significa incidencia, campaña esperada o cambio de población?
Evidencia de release	CI, tests, artefactos versionados	Bloquear o aprobar ventanas con pruebas reproducibles.	¿El gate deja una decisión auditable o solo un número bonito?

Para un proyecto pequeño, un `Makefile`, scripts Python y JSON bien escritos pueden enseñar mejor que una plataforma enorme. Para producción, la pregunta cambia: cómo se integra con CI/CD, dónde viven los secretos, quién firma el cambio de contrato, cuánto cuesta retener trazas, qué datos personales viajan en logs y cómo se versiona cada artefacto. El cuaderno del facsímil usa un montaje mínimo para que se entienda el mecanismo antes de subirlo a una plataforma real.

Contratos entre servicios y evolución de schema

En capítulos anteriores hablamos de contrato de datos. Aquí ampliamos la idea: un pipeline de producción tiene contratos entre productores y consumidores. El productor promete columnas, tipos, semántica, frecuencia y trazabilidad. El consumidor promete cómo usará esos campos y qué cambios acepta.

dbt model contracts permiten declarar y hacer cumplir columnas, tipos y restricciones en modelos dbt.[1][2] Great Expectations organiza expectativas verificables sobre datos. OpenLineage conecta jobs y datasets. OpenTelemetry conecta ejecución y trazas. Son capas distintas de la misma idea: **un cambio no debería romper al consumidor en silencio**.

CAMBIO	COMPATIBLE	REQUIERE REVISIÓN	ROMPE
Añadir columna opcional	Sí	Si afecta coste o privacidad.	No.
Añadir valor nuevo a catálogo	A veces.	Sí, si cambia slices o política.	Si el consumidor no lo acepta.
Renombrar columna obligatoria	No.	Sí.	Sí, salvo alias versionado.
Cambiar significado de `decision`	No.	Sí.	Sí.
Quitar `trace_id`	No.	Sí.	Sí.
Subir un SLO	Sí.	Sí, si bloquea más ventanas.	No necesariamente.

Un contrato de evolución debería declarar:

PIEZA	EJEMPLO
Compatibilidad hacia atrás	La versión nueva acepta datos de la versión anterior.
Compatibilidad hacia delante	La versión anterior puede ignorar campos nuevos opcionales.
Deprecación	Campo permitido hasta una fecha o versión.
Aliases	`student_profile` puede aceptar temporalmente `profile`.
Campo obligatorio	`trace_id` no se puede quitar.
Política de rollback	Volver a `pipe-1.4.1` conserva outputs y hashes.

SLIs de datos que no son drift

Drift es importante, pero no es el único síntoma que rompe un sistema de IA. Wang y Strong ya separaban la calidad de datos en dimensiones que importan a los consumidores del dato, no solo al productor: exactitud, completitud, actualidad, relevancia, interpretabilidad y accesibilidad, entre otras.[13] En ingeniería de IA eso se traduce en una idea sencilla: una ventana puede no tener drift fuerte y aun así no ser usable porque llegó tarde, incompleta, sin etiqueta, sin trazas o con un contrato parcialmente roto.

Un SLI de datos debe tener tres piezas: una unidad medible, una ventana temporal y una consecuencia. Si dices “calidad de datos”, todavía no has dicho nada operativo. Si dices “porcentaje de eventos de decisión recibidos antes de 15 minutos desde su creación, por ventana y por canal”, ya puedes medir, alertar y decidir.

SLI DE DATOS	QUÉ MIDE	EJEMPLO DE CÁLCULO OPERATIVO	CUÁNDO BLOQUEA
Freshness	Edad del dato cuando llega al pipeline.	`event_available_at - event_created_at` .	Si el dato llega después de que la decisión ya se haya tomado.
Completitud	Campos obligatorios presentes.	`filas_con_campos_obligatorios / filas_totales` .	Si falta `trace_id` , `event_id` , `decision` , `window` o campo de slice crítico.
Validez	Valores dentro del contrato.	`valores_validos / valores_revisados` .	Si aparece un catálogo no permitido y el consumidor no sabe interpretarlo.
Unicidad	Duplicados por clave de idempotencia.	`duplicados(window,event_id)` .	Si un retry puede duplicar decisiones, alertas o revisiones humanas.
Puntualidad	Si la ventana cierra a tiempo.	`window_closed_at <= deadline` .	Si el batch llega tarde y contamina informes o entrenamiento.
Label delay	Retraso de la etiqueta real.	`label_available_at - decision_at` .	Si se evalúa miss rate antes de que haya verdad de terreno suficiente.
Trazabilidad	Eventos con traza investigable.	`eventos_con_trace_id / eventos_totales` .	Si una ventana no se puede depurar cuando falla.
Cobertura de slices	Slices críticos presentes con muestra mínima.	`n(slice) >= mínimo` .	Si una media global oculta que el segmento importante no llegó.

El valor de esta tabla está en obligarnos a separar síntomas. Freshness contesta si el dato llega a tiempo. Completitud contesta si llega entero. Validez contesta si respeta el contrato. Label delay contesta si estamos evaluando antes de que exista la verdad de terreno. Trazabilidad contesta si podremos investigar. Son dimensiones distintas; mezclarlas bajo la palabra “calidad” hace que el equipo reaccione tarde y mal.

Great Expectations y Deequ existen precisamente para expresar checks de calidad como reglas reproducibles en vez de como revisión manual difusa.[14][15] La parte que no puede hacer la herramienta por ti es decidir qué SLI tiene consecuencia. En un producto educativo, `label_delay` puede impedir evaluar una política durante varios días. En un RAG, `freshness` puede significar que una normativa nueva no entró al índice. En un sistema con agentes, `completitud` puede ser que falta el resultado de una herramienta y el agente está razonando con media verdad.

Para aterrizarlo: si una universidad recibe solicitudes de beca, una ventana con todos los campos completos pero con etiquetas reales retrasadas no sirve para estimar `miss_rate` de forma honesta. Puedes medir latencia, drift y trazabilidad, pero no puedes cerrar evaluación de acierto hasta que llegue la resolución real. En cambio, si el problema es freshness, quizá sí tienes etiquetas, pero llegaron demasiado tarde para que la alerta protegiera la decisión. Son fallos distintos y piden defensas distintas.

CASO REALISTA	SLI QUE MANDA	ACCIÓN RAZONABLE
Documentos de políticas actualizados por la noche.	Freshness del índice documental.	No usar el índice para respuestas normativas hasta terminar ingestión y validación.
Tickets con `access_need` vacío por cambio de formulario.	Completitud por slice crítico.	Bloquear automatización y corregir el productor del evento.
Etiquetas de resolución llegan 10 días después.	Label delay.	Separar monitorización temprana de evaluación final con etiqueta.
Reintento de pipeline repite `event_id`.	Unicidad por clave idempotente.	Detener escritura de salida y exigir replay controlado.
Aparece `language=pt` sin contrato.	Validez de catálogo.	Enviar a `review` hasta versionar contrato, slices y evaluación.

Observabilidad correlacionada

La observabilidad de IA no consiste en guardar un log grande. Consiste en poder saltar de una alerta a los eventos concretos y de ahí a la traza que explica dónde se degradó.

Una traza mínima de decisión debería poder responder:

CAMPO	POR QUÉ IMPORTA
`trace_id`	Une logs, métricas, spans y evento final.
`event_id`	Identifica la unidad de decisión.
`pipeline_version`	Distingue cambio de código de cambio de datos.
`model_version`	Permite comparar comportamiento por versión.
`data_version`	Evita investigar con otro snapshot.
`policy_version`	Explica por qué el mismo score acabó en otra decisión.
`span_name`	Localiza ingesta, validación, scoring, decisión o emisión.
`duration_ms`	Separa problema de calidad de problema de latencia.

En el cuaderno del facsímil, el contrato de ingeniería exige spans `ingest`, `validate`, `score`, `decide` y `emit`. El evento `p013` no tiene `trace_id`. El evento `p017` tiene traza, pero le falta `emit`. Los eventos `p011`, `p012` y `p017` muestran `score` lento. Esa correlación permite decir algo concreto: la ventana `2026-06-08` no solo cambió de distribución; también llegó peor instrumentada y más lenta.

EVENTO	VENTANA	SEÑAL
`p011`	`2026-06-08`	Traza total `710 ms`, `score` lento.
`p012`	`2026-06-08`	Traza total `755 ms`, `score` lento.
`p013`	`2026-06-08`	Sin `trace_id`.
`p017`	`2026-06-08`	Falta span `emit`, `score` lento.

Sin esta correlación, el equipo podría culpar al modelo, al dato o al usuario sin evidencia suficiente. Con trazas, la pregunta cambia: ¿qué parte del pipeline cambió entre `pipe-1.4.1` y `pipe-1.4.2`?

Esta es una diferencia profunda con una cultura de logs sin diseño. Un log suelto dice que algo ocurrió; una traza correlacionada permite reconstruir una decisión. Si `event_id`, `trace_id`, versión de pipeline, versión de datos y spans no viajan juntos, cada dashboard cuenta una parte de la historia. DataOps consiste en que esas partes se puedan juntar cuando más falta hace.

Telemetría sin datos sensibles

Hay una tensión real: para investigar necesitas contexto, pero si conviertes logs y trazas en un vertedero de datos personales, has creado otro problema. El Reglamento General de Protección de Datos exige principios como minimización, limitación de finalidad y protección desde el diseño; NIST Privacy Framework propone gestionar privacidad como riesgo de ingeniería, no como nota legal al final.[16][17]

En DataOps para IA esto significa que `trace_id` debe existir, pero no debe ser el DNI del usuario, el email del alumno ni el texto completo de una consulta sensible. Una traza útil necesita correlación, versiones, spans, duración y estado. No necesita copiar el contenido completo del documento privado que pasó por el pipeline.

ELEMENTO DE TELEMETRÍA	ÚTIL	RIESGO	FORMA MÁS SEGURA
`trace_id`	Correlacionar eventos.	Reidentificar si contiene datos personales.	ID aleatorio o hash no reversible con sal.
`user_email`	Depurar un caso concreto.	PII directa en logs.	Pseudónimo interno o referencia a sistema con permisos.
Texto completo del prompt	Diagnosticar comportamiento.	Puede contener datos personales, secretos o documentos privados.	Guardar hash, plantilla, longitud, categoría y muestra redaccionada si hay permiso.
Labels de Prometheus	Filtrar métricas por dimensión.	Alta cardinalidad y fuga de identificadores.	Labels de baja cardinalidad: modelo, versión, canal, región, slice permitido.
Span attributes	Entender paso lento o fallido.	Meter payloads enteros en trazas.	Estado, duración, versión, error class y tamaños, no contenido sensible.
Retención	Investigar incidentes.	Acumular más datos de los necesarios.	TTL por tipo de señal y política de borrado.

La observabilidad útil no debería convertirse en una copia paralela de producción llena de datos sensibles. Este equilibrio es difícil, pero muy práctico: guardar identificadores técnicos, hashes, tamaños, versiones y clases de error suele bastar para detectar patrones; revisar contenido completo debería ser un flujo excepcional, con permisos, redacción y auditoría. Si no diseñamos esto desde el principio, la depuración se vuelve cómoda a costa de privacidad y riesgo.

Prometheus recomienda nombres y etiquetas consistentes; en la práctica, además, conviene evitar labels de cardinalidad explosiva como `user_id`, `document_id`, `email` o `raw_prompt`, porque rompen coste, rendimiento y privacidad.[18] OpenTelemetry permite enriquecer trazas

con atributos, pero el criterio de ingeniería es el mismo: atributo sí, volcado indiscriminado no. Una traza sana te permite saber que `score` tardó 420 ms en `pipe-1.4.2` para `language=en`; no debería obligarte a guardar el texto personal de la solicitud.

Ejemplo concreto: para depurar un RAG académico no necesitas registrar la pregunta completa “mi expediente con DNI X tiene…” ni el PDF privado usado como contexto. Puedes registrar `retrieval_k=8`, `top_source_type=normativa`, `doc_version=2026-06-01`, `query_hash`, `prompt_template_version`, `input_token_count`, `output_token_count`, `policy_version` y `trace_id`. Si hace falta revisar contenido, se abre un flujo con permisos, redacción y auditoría. Eso es más lento que guardar todo, sí. También es la diferencia entre observabilidad profesional y una fuga esperando fecha.

Testing de pipelines de IA

Un pipeline de IA necesita tests, pero no todos los tests son iguales. Un test unitario puede comprobar que una función calcula una media. Eso es necesario, pero insuficiente. En producción también hay que probar contratos, ventanas, trazabilidad, linaje, regresiones por slices y comportamiento operativo.

TIPO DE TEST	QUÉ COMPRUEBA	EJEMPLO ÚTIL
Unitario	Una función aislada.	`total_variation_distance(P, Q)` devuelve `0.8` en un caso conocido.
Contract test	Productor y consumidor hablan el mismo idioma.	`trace_id`, `event_id`, `decision`, `latency_ms` y `window` existen y tienen tipo esperado.
Data quality test	La ventana tiene datos válidos.	No hay nulos en campos obligatorios, catálogos permitidos y rangos razonables.
Lineage test	La run conserva procedencia.	Los hashes de inputs y outputs quedan escritos en `lineage_event.json`.
Regression test	Una versión nueva no empeora una referencia.	`pipe-1.4.2` no reduce captura segura frente a `pipe-1.4.1`.
Slice test	No se esconde un fallo en la media.	`language=en` y `access_need=si` mantienen SLO propio.
Trace test	La operación se puede investigar.	Cada evento crítico tiene spans `ingest`, `validate`, `score`, `decide` y `emit`.
Replay test	Se puede reproducir una ventana.	Reejecutar `2026-06-08` con la misma versión produce la misma decisión.

El error típico es probar solo el código y no probar el sistema. En IA, el sistema incluye datos, contratos, scheduler, runtime, versión de modelo, política de decisión, evals, slices y observabilidad. Por eso el ML Test Score no se limita a accuracy: pregunta por tests de datos, monitorización, cambios y dependencia de features.[19]

Para un equipo de ingeniería, una regla práctica sería esta:

```
No hay release de pipeline si no pasan:
1. contrato de datos,
2. contrato de trazas,
3. replay de una ventana conocida,
4. scorecard de slices críticos,
5. evento de linaje con hashes.
```

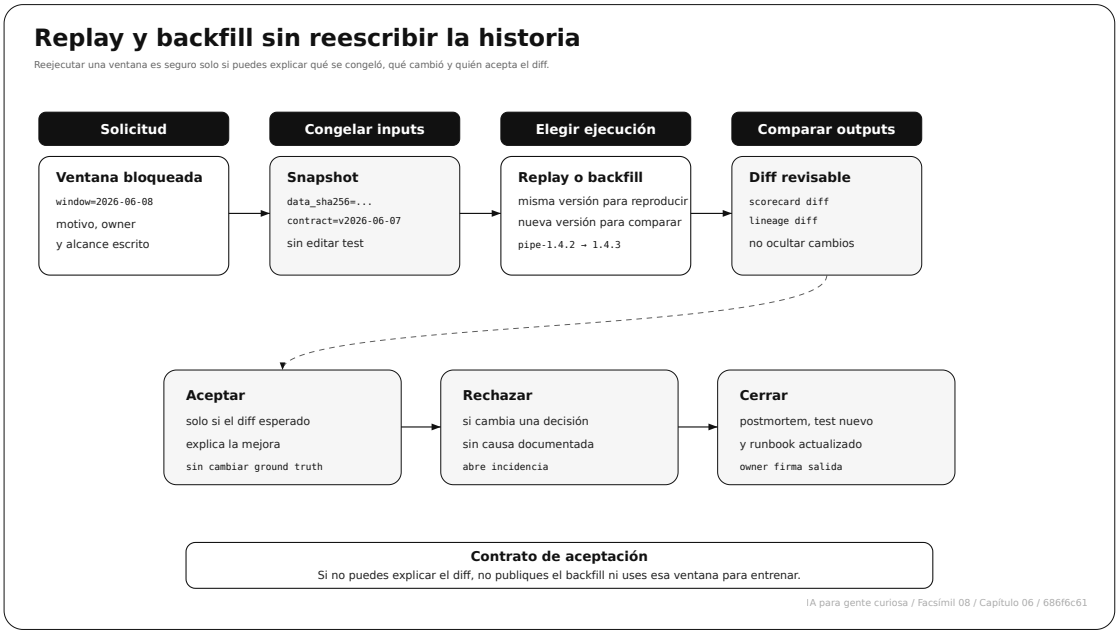
Esto parece mucho hasta que ocurre la primera incidencia real. Entonces descubres que esos cinco puntos no son burocracia: son la diferencia entre investigar en una hora o discutir durante días.

Backfill y replay sin romper histórico

Backfill y replay suelen parecer palabras de plataforma, pero para IA son decisiones delicadas. Un **replay** reejecuta una ventana para reproducir o comparar una decisión. Un **backfill** recalcula ventanas antiguas para completar histórico o aplicar una versión nueva. Los dos pueden ser sanos; los dos pueden destruir trazabilidad si se hacen sin contrato.

La regla profesional es esta: no se recalcula histórico “porque sí”. Se abre un expediente de backfill. Ese expediente dice qué ventana se toca, qué snapshot entra, qué código se usa, qué contrato gobierna, qué política de decisión se aplica, qué outputs se esperan, qué diff se acepta y qué queda prohibido. Google SRE trata los cambios operativos e incidentes como sistemas que necesitan preparación, respuesta y aprendizaje; en IA aplicada, backfill y replay son parte de esa disciplina.[20][21]

PASO	QUÉ SE CONGELA O REVISA	EJEMPLO
1. Motivo	Por qué se reejecuta.	`trace_id` faltante en ventana `2026-06-08`.
2. Snapshot	Datos exactos de entrada.	`production_events.csv` con hash SHA-256.
3. Código	Versión de pipeline.	`pipe-1.4.2` o parche `pipe-1.4.3`.
4. Contrato	Reglas vigentes.	`monitoring_contract.json` versión fechada.
5. Política	Umbrales y gates.	No cambiar SLOs para que el backfill “pase”.
6. Salida esperada	Artefactos a comparar.	`monitoring_report.json`, `slo_scorecard.csv`, `lineage_event.json`.
7. Diff permitido	Cambios aceptables.	Se corrige traza; no cambia etiqueta ni decisión sin justificación.
8. Cierre	Quién firma y qué test queda.	Postmortem + test de spans obligatorios.



> Replay y backfill necesitan un contrato propio: snapshot, versión, política, diff aceptable, postmortem y test nuevo.

Ejemplo: si el evento `p017` no emitió el span `emit`, no debes “rellenar” el histórico a mano. Primero congelas la ventana, reejecutas con la misma versión para comprobar reproducibilidad y luego pruebas el parche que fuerza spans obligatorios. Si el único cambio

esperado es que `p017` ahora tiene traza completa y el resto del scorecard no cambia, el backfill es defendible. Si cambian decisiones, etiquetas o slices sin explicación, no es un backfill: es una alteración del experimento.

Drift, SLI, SLO y presupuesto de error

El drift se mide comparando una distribución de referencia P con una distribución actual Q. Una medida clásica es la distancia de variación total, habitual en probabilidad y teoría de la información.[22]

$$TV(P, Q) = \frac{1}{2} \sum_{c \in C} |P(c) - Q(c)|$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
P	Distribución de referencia.	Idiomas durante validación.
Q	Distribución actual.	Idiomas del 8 de junio de 2026.
C	Categorías posibles.	`es`, `ca`, `en`.
$P(c)$	Proporción de la categoría c en referencia.	`en = 0.2`.
$Q(c)$	Proporción de la categoría c en producción.	`en = 1.0`.
$TV(P, Q)$	Distancia entre 0 y 1.	`0.8`, cambio fuerte.

En palabras: $TV(P, Q)$ suma cuánto se mueve cada categoría entre referencia y producción y divide por dos para no contar el desplazamiento dos veces. Si en referencia el idioma `en` era el 20 % y en producción pasa al 100 %, mientras `es` y `ca` desaparecen de esa ventana, el cambio es enorme. La fórmula no dice por sí sola “incidencia”: dice que la ventana ya no se parece a la referencia y que debes abrir una investigación.

También se usa PSI (*Population Stability Index*) en scoring, riesgo y monitorización de distribuciones. Es una señal industrial útil, no una prueba universal.[23]

$$PSI(P, Q) = \sum_{c \in C} (Q(c) - P(c)) \log \frac{Q(c)}{P(c)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
PSI	Índice de estabilidad poblacional.	`11.711575` para `language`.
$\log$	Logaritmo natural.	Penaliza cambios grandes de proporción.
$Q(c) - P(c)$	Diferencia de proporciones.	`en` sube mucho.

En palabras: PSI pesa la diferencia de proporciones por el logaritmo del cociente entre producción y referencia. Por eso castiga mucho los casos donde una categoría casi desaparece o aparece de golpe. En ingeniería se usa como señal de estabilidad, no como verdad absoluta. Si `language` tiene PSI altísimo, no se reentrena automáticamente: primero se pregunta si hubo campaña, cambio de canal, bug de ingestión, cambio de producto o sesgo de muestra.

Pero drift no es lo único. Para operar necesitamos SLIs y SLOs.

CONCEPTO	DEFINICIÓN	EJEMPLO
SLI	Indicador medible.	`latency_p95_ms`, `miss_rate`, `missing_trace_rate`.
SLO	Objetivo sobre un SLI.	`latency_p95_ms <= 650`.
Presupuesto de error	Margen tolerado antes de frenar cambios.	Si `miss_rate > 0.12`, no se aumenta automatización.
Gate	Regla que convierte SLOs en estado.	`block` si falta trazabilidad o cae captura segura.

Prometheus insiste en que los nombres de métricas deben ser claros, consistentes y expresar unidades cuando aplica.[24] Eso parece menor hasta que tienes que investigar una alerta a las 9 de la mañana. `latency` es ambiguo; `decision_latency_p95_ms` dice mucho más.

Presupuesto consumido y burn rate

En ingeniería de fiabilidad, el presupuesto de error traduce un SLO a margen consumible: si prometes que una proporción mínima de eventos cumple, la parte que puede fallar antes de frenar cambios es $1 - s$, donde s es el objetivo del SLO. Google SRE popularizó esta forma de conectar señales operativas con riesgo para usuarios.[25] El *SRE Workbook* lo lleva a alerting basado en consumo de presupuesto, no solo en umbrales aislados.[26]

En un capítulo de datos para IA, esta idea no se limita a disponibilidad web. Puede aplicarse a una ventana de decisiones: eventos sin traza, casos críticos no capturados, spans que superan latencia, registros fuera de contrato o slices que incumplen el mínimo pactado. Lo importante es no llamarlo “intuición”. Debe quedar escrito qué cuenta como evento malo, cuál era el SLO y qué ocurre si se consume demasiado presupuesto.

Si una ventana contiene N eventos evaluables y el SLO exige una proporción mínima s de eventos correctos, el presupuesto de error permitido para esa ventana es:

$$B = (1 - s)N$$

SÍMBOLO	SIGNIFICADO	EJEMPLO DEL CUADERNO
B	Número esperado de errores permitidos antes de gastar el presupuesto de la ventana.	`1.2` fallos permitidos.
s	Objetivo del SLO expresado como proporción.	`0.88` de captura segura mínima.
N	Eventos evaluables en la ventana.	`10` eventos.

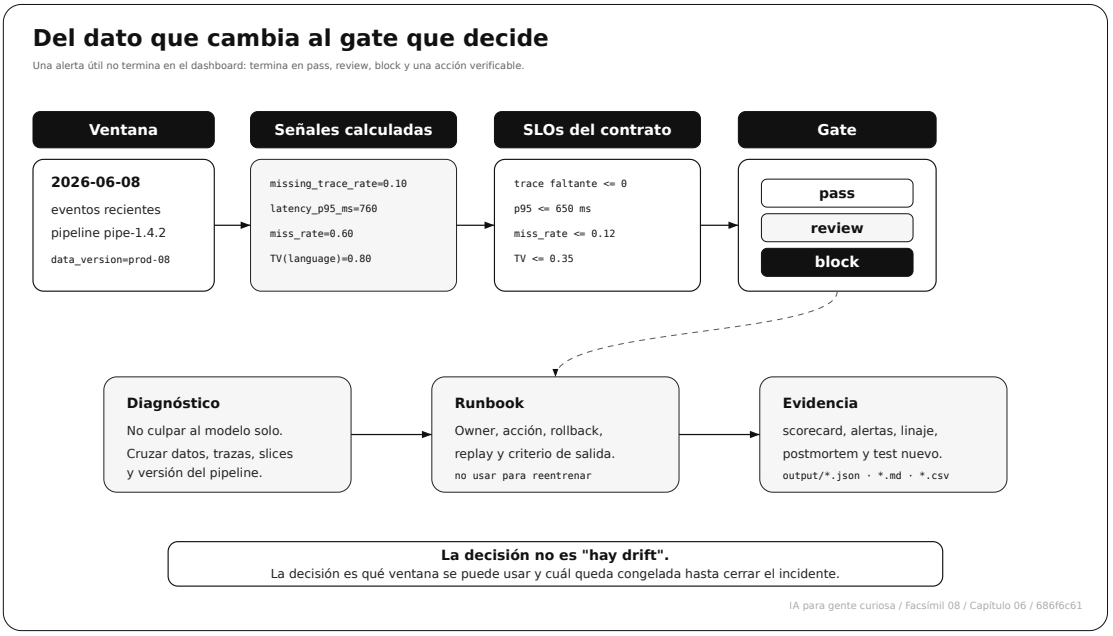
En palabras: si aceptas como SLO que al menos el 88 % de los casos críticos queden capturados, estás aceptando como margen máximo un 12 % de fallos. En una ventana de 10 eventos, eso equivale a 1.2 fallos permitidos. No significa que puedas tener “un fallo con decimales”; significa que, agregando ventanas, ese es el margen estadístico que estás consumiendo. Si observas 6 fallos, la ventana no está cerca del límite: lo ha atravesado con claridad.

Para comparar ventanas de distinto tamaño se usa la idea de *burn rate*: cuántas veces más rápido estás consumiendo el presupuesto de error respecto a lo permitido por el SLO. En una ventana w, con e_w eventos malos entre n_w eventos evaluables:

$$BR_w = \frac{e_w / n_w}{1 - s}$$

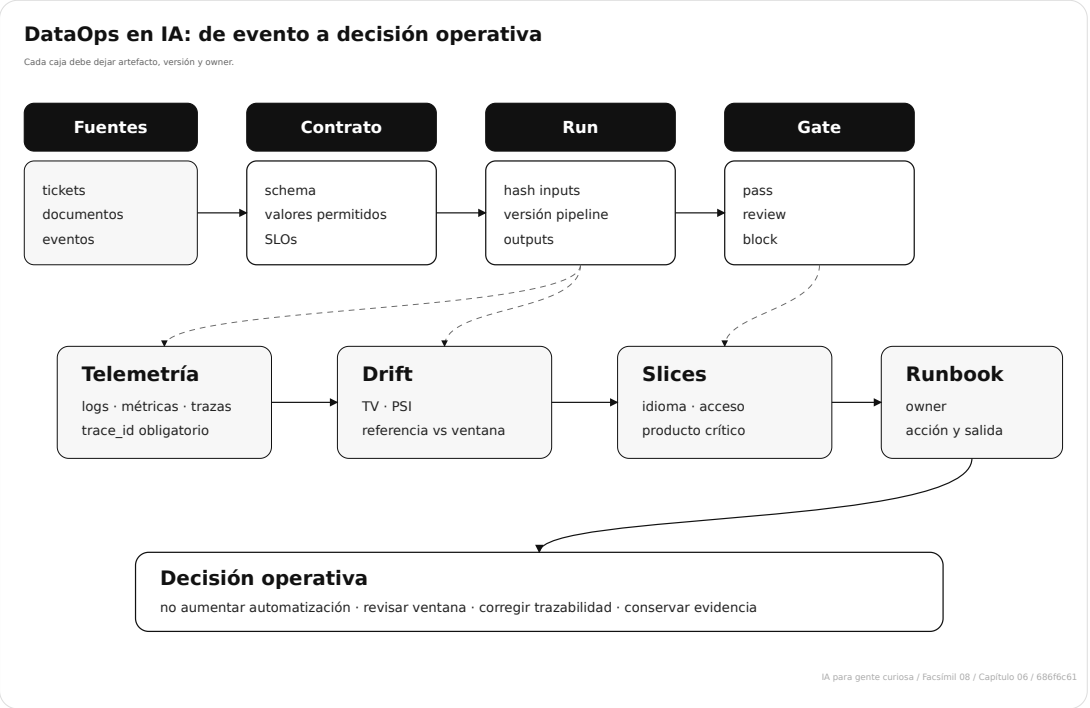
SÍMBOLO	SIGNIFICADO	EJEMPLO DEL CUADERNO
BR_w	Burn rate de la ventana w.	`5.0` , consumo cinco veces superior al permitido.
e_w	Eventos malos observados en la ventana.	`6` fallos.
n_w	Eventos evaluables en la ventana.	`10` eventos.
$1 - s$	Fracción de error permitida por el SLO.	`0.12` .

En palabras: si el SLO permite fallar un 12 % y la ventana falla un 60 %, el burn rate es $0.60/0.12 = 5$. Esa cifra ya no es una opinión sobre si “parece mal”: es una señal operativa para bloquear, abrir incidente y exigir replay antes de reusar la ventana. En sistemas de IA esto importa muchísimo porque una ventana rota puede acabar convertida en dataset de reentrenamiento, benchmark interno o decisión de producto.



El gate convierte señales sueltas en una decisión operativa: qué se bloquea, quién actúa y qué evidencia queda.

Arquitectura operativa de un sistema de datos para IA



Un pipeline DataOps convierte eventos en evidencia: contrato, run, telemetría, drift, slices, gate y runbook.

En el día a día

En producción trabajamos con ventanas. Una ventana puede ser un día, una hora, un batch, una versión de modelo, una campaña o un segmento de tráfico. La ventana es importante porque las medias largas esconden problemas cortos.

En el cuaderno del facsímil, la ventana `2026-06-07` pasa. La ventana `2026-06-08` bloquea.

VENTANA	ESTADO	N	TRACE FALTANTE	LATENCIA P95	REVISIÓN	PÉRDIDA	CAPTURA SEGURA	FLAGS
2026-06-07	`pass`	10	0.0	590.0	0.4	0.0	1.0	0
2026-06-08	`block`	10	0.1	760.0	0.7	0.6	0.4	15

La segunda ventana no bloquea por una sola razón. Bloquea porque varias señales coinciden:

SEÑAL	LECTURA
`missing_trace_rate = 0.1`	Falta trazabilidad; una investigación quedaría incompleta.
`latency_p95_ms = 760`	La ventana supera el SLO de latencia.
`miss_rate = 0.6`	Demasiados casos prioritarios pasan a flujo normal.
`safety_capture = 0.4`	La política ya no captura de forma segura los casos importantes.
Drift en `language` y `access_need`	Producción se aleja mucho de la referencia.
Slices críticos fallan	`language=en` , `access_need=si` y `product=practicas` concentran señales.

Aquí se ve la diferencia entre dashboard y operación. Un dashboard muestra números. Un gate operativo dice: **no aumentes automatización ni uses esta ventana para reentrenar sin investigar**.

Severidad: no todas las alertas pesan igual

Un capítulo de DataOps necesita una matriz de severidad porque no todo `review` ni todo `block` significan lo mismo. Una alerta de latencia aislada puede pedir seguimiento. Una ventana sin trazas en eventos críticos puede impedir investigar una decisión. Un slice crítico con `miss_rate` alto puede afectar directamente a usuarios. Google SRE separa respuesta de emergencia, gestión de incidentes y cultura de postmortems para evitar que todo se trate igual y para que cada incidente deje aprendizaje verificable.[272829]

Para IA, la severidad no se decide solo por cuántas filas fallan. Se decide por consecuencia: si afecta una decisión automática, si contamina entrenamiento, si rompe trazabilidad, si toca un slice crítico, si expone datos sensibles o si impide medir el resultado real.

SEVERIDAD	SEÑAL TÍPICA	CONSECUENCIA	ACCIÓN MÍNIMA
S0	Exposición de datos sensibles en logs, decisión automática dañina o pérdida masiva de trazabilidad.	Riesgo para personas, cumplimiento o continuidad.	Congelar ventana, parar automatización afectada, abrir incidente y activar responsable.
S1	Slice crítico con `block`, `trace_id` faltante en eventos críticos o miss rate muy por encima del SLO.	No se puede defender la ventana.	No reentrenar, no aumentar automatización, replay y postmortem.
S2	Drift fuerte, latencia alta o revisión humana por encima de capacidad.	La ventana puede operar con restricciones, pero no promocionarse.	`review`, owner, seguimiento y acción correctiva fechada.
S3	Alerta leve sin impacto directo y con trazabilidad completa.	Señal de observación.	Registrar, mirar tendencia y no abrir incidente mayor.

Ejemplo: `latency_p95_ms = 760` puede ser S2 si la decisión sigue siendo correcta y trazable. `missing_trace_rate = 0.1` puede subir a S1 si afecta eventos críticos, porque sin traza no puedes reconstruir qué pasó. Una fuga de prompt con datos personales en logs no es “solo observabilidad”: puede ser S0 aunque el modelo responda bien.

La matriz de severidad evita dos errores opuestos. El primero es dramatizar todo y saturar al equipo. El segundo es normalizar señales graves porque el dashboard tiene muchas luces. Un sistema maduro no reacciona por ansiedad: reacciona porque una regla escrita conecta señal, consecuencia y acción.

Incidentes, postmortems y acciones correctivas

Cuando una ventana queda en `block`, el trabajo no termina en “hay alerta”. Empieza la parte profesional: reconstruir impacto, señales, causa probable, acción correctiva y criterio de cierre. El postmortem no busca repartir culpa. Busca impedir que el mismo patrón vuelva a repetirse sin que el sistema lo detecte.

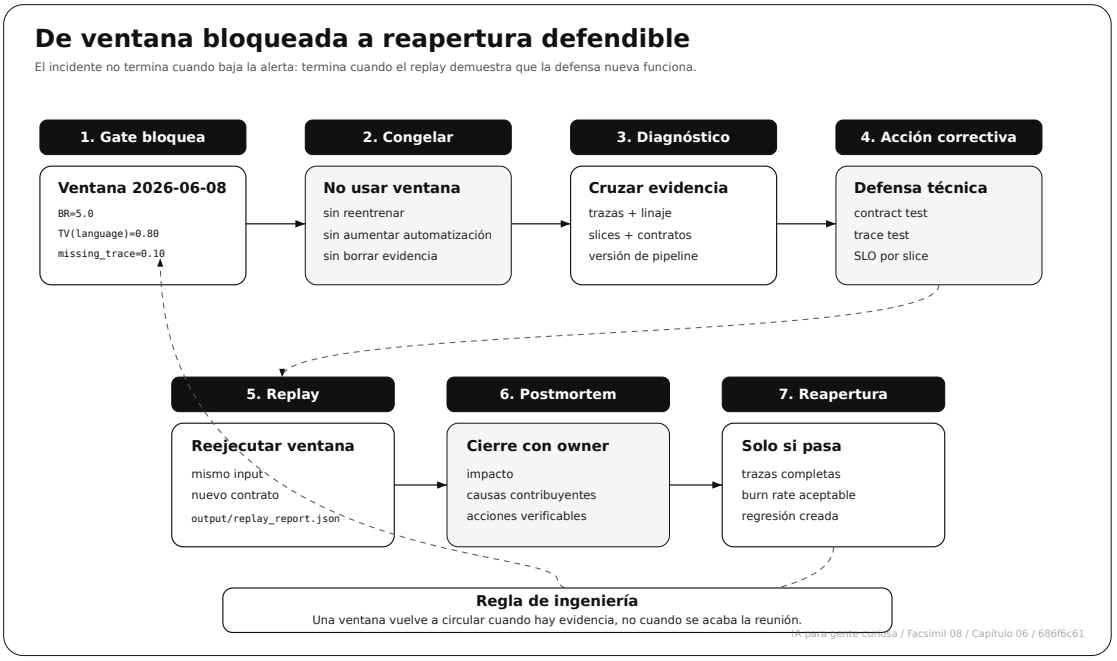
Un postmortem técnico mínimo debería incluir:

BLOQUE	PREGUNTA QUE RESPONDE	EJEMPLO DEL CUADERNO
Resumen	¿Qué estado queda y qué se bloquea?	`Estado de ingeniería: block`.
Impacto	¿Qué decisión queda afectada?	No aumentar automatización en `2026-06-08`.
Timeline	¿Qué scripts y señales detectaron el problema?	`monitor_dataops.py` y `inspect_pipeline_engineering.py`.
Señales	¿Qué eventos concretos fallaron?	`p013` sin `trace_id`, `p017` sin `emit`.
Causa probable	¿Qué explicación encaja con la evidencia?	Cambio de distribución más trazabilidad incompleta y `score` lento.
Acción correctiva	¿Qué se cambia en código, contrato o operación?	Hacer `trace_id` bloqueante y añadir test de spans.
Criterio de cierre	¿Cómo sabemos que está resuelto?	Replay con trazas completas y scorecard sin `block`.

El detalle importante es que cada acción correctiva debe convertirse en una defensa técnica. Si el problema fue una columna obligatoria ausente, añade un contract test. Si fue una traza incompleta, añade un trace test. Si fue un retry que duplicó efectos, añade clave de idempotencia. Si fue una ventana de datos rota, añade un gate que impida usarla para reentrenamiento.

PROBLEMA DETECTADO	ACCIÓN POBRE	ACCIÓN DE INGENIERÍA
Falta `trace_id`	“Mirarlo manualmente”.	Bloquear emisión si falta `trace_id` en eventos críticos.
Span `score` lento	“Optimizar algo”.	Medir p95 por versión y crear SLO por span.
Slice crítico degradado	“Avisar al equipo”.	Añadir SLO por slice y replay de la ventana.
Drift fuerte	“Reentrenar rápido”.	Separar drift esperado, drift operativo y datos no aptos.
Backfill de histórico	“Recalcular y ya”.	Congelar versión de código, contrato, datos y política.

Un postmortem útil también deja una decisión negativa explícita: **qué no se va a hacer**. En el cuaderno del facsímil, no se usa `2026-06-08` para aumentar automatización ni para reentrenar hasta que el gate cierre. Esa frase protege al equipo de convertir datos dudosos en verdad operativa.



El ciclo profesional de una ventana bloqueada: congelar, diagnosticar, corregir, reproducir y reabrir solo con evidencia.

Por qué debería importarte

Si no entiendes DataOps, puedes tomar malas decisiones con muy buena intención. Puedes reentrenar con una ventana rota. Puedes comparar modelos con datos que ya no representan producción. Puedes ignorar un slice crítico porque la media global parece estable. Puedes perder trazas justo cuando más las necesitas.

La operación de IA necesita una idea incómoda: el modelo no vive solo. Vive dentro de contratos, datos, versiones, colas, revisiones humanas, latencia, permisos, trazas y decisiones de producto.

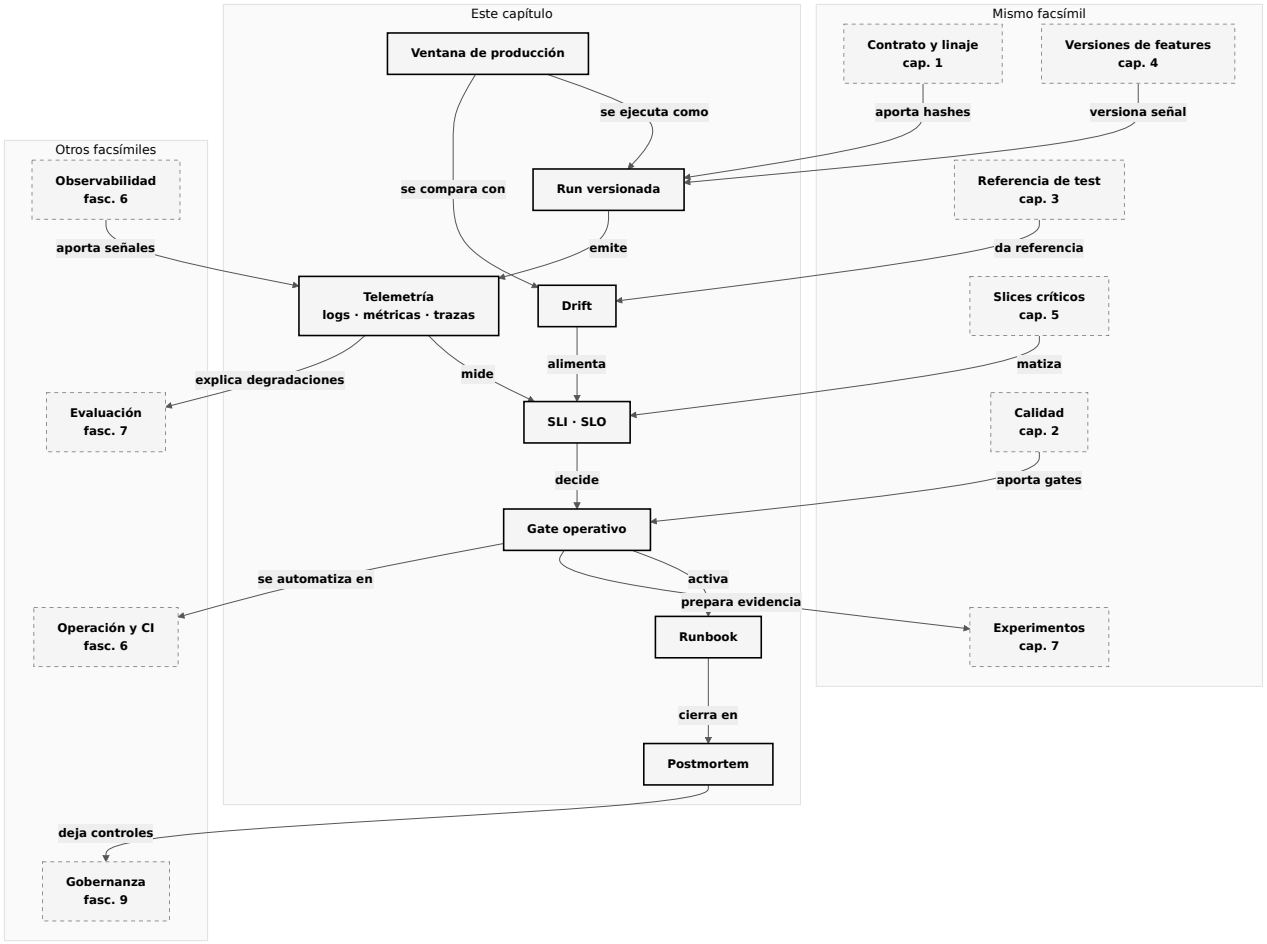
Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Mirar solo métricas globales	La media da sensación de control.	Medir ventanas y slices críticos.
Confundir drift con fallo automático	Un cambio puede ser esperado.	Revisar causa, contexto y consecuencia.
No exigir `trace_id`	Parece detalle de logging.	Tratar trazabilidad como SLO bloqueante.
Tener alerta sin runbook	El dashboard avisa, pero nadie sabe qué hacer.	Escribir owner, acción y criterio de salida.
Reentrenar con producción rota	Se intenta arreglar rápido.	Bloquear ventanas con gate `block`.
Reintentar sin idempotencia	Se piensa que retry siempre es inocente.	Definir `idempotency_key` antes de automatizar retries.
Hacer backfill sin congelar versión	Parece solo recalcular.	Guardar datos, código, contrato, política y hashes.
Escribir postmortem sin test nuevo	Se documenta el problema, pero no se evita repetirlo.	Cada postmortem debe cerrar con una defensa verificable.

Cómo encaja todo

Este mapa se lee como continuidad del facsímil. Los capítulos 01 a 05 construyeron contrato, calidad, split, representación y slices. Este capítulo convierte esas piezas en operación: ventanas, SLIs, SLOs, drift, trazas, linaje y runbooks.

La decisión que enseña no es “qué modelo elegir”. Es más básica y más profesional: **qué ventana de producción es apta para seguir tomando decisiones y cuál exige revisión**. Después, el capítulo 07 usará esta evidencia para análisis aplicado, experimentos y causalidad.


Este capítulo
Mismo facsímil
Otros facsímiles
Ventana de producción
se ejecuta como
Run versionada
se compara con
emite
Drift
Telemetría
logs · métricas · trazas
alimenta
mide
explica degradaciones
SLI · SLO
decide
Gate operativo
se automatiza en
activa
prepara evidencia
Runbook
cierra en
Postmortem
deja controles
Contrato y linaje
cap. 1
Versiones de features
cap. 4
aporta hashes
versiona señal
Referencia de test
cap. 3
da referencia
Slices críticos
cap. 5
matiza
Calidad
cap. 2
aporta gates
Experimentos
cap. 7
Observabilidad
fasc. 6
aporta señales
Evaluación
fasc. 7
Operación y CI
fasc. 6
Gobernanza
fasc. 9

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
DataOps	Operar datos como producto versionado, verificable y monitorizable.
Pipeline	Secuencia reproducible de pasos que transforma entradas en artefactos.
Run	Ejecución concreta con inputs, outputs, versiones, hashes y estado.
Drift	Cambio medible entre referencia y producción.
SLI	Indicador medible.
SLO	Objetivo sobre un SLI.
Presupuesto de error	Margen permitido antes de frenar cambios.
Burn rate	Velocidad relativa a la que una ventana consume el presupuesto de error.
Trace ID	Identificador que permite seguir una operación por el sistema.
Lineage event	Evento que registra job, run, entradas, salidas, hashes y relación entre datasets.
Freshness	Edad operativa del dato respecto al momento en que se necesita para decidir.
Completitud	Proporción de campos, eventos o relaciones obligatorias que llegan con valor usable.
Puntualidad	Capacidad de que el dato llegue antes del cierre operativo de su ventana.
Label delay	Retraso entre una predicción y la etiqueta real que permite evaluarla.
Cardinalidad	Número de valores distintos de una etiqueta, campo o dimensión en métricas, logs o trazas.
Minimización de datos	Recoger y conservar solo los datos necesarios para una finalidad concreta.
Severidad	Clasificación que conecta señal, consecuencia, urgencia y acción mínima.
Runbook	Guía de acción cuando una alerta o gate falla.
Gate operativo	Regla que convierte señales en `pass`, `review` o `block`.
Idempotencia	Propiedad que permite repetir una operación sin duplicar efectos.

TÉRMINO	DEFINICIÓN BREVE
Backfill	Reprocesado de ventanas antiguas con versión controlada de datos, código y contrato.
Replay	Reejecución de una ventana para reproducir o comparar una decisión.
Span	Tramo de una traza que representa un paso medible de una operación.
Contract test	Test que verifica que productor y consumidor cumplen el contrato pactado.
Postmortem	Documento técnico que convierte una incidencia en acciones verificables.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Qué diferencia hay entre pipeline y run?
2. ¿Por qué una ventana corta puede contar más que una media larga?
3. ¿Qué mide `missing_trace_rate` y por qué puede bloquear?
4. ¿Qué diferencia hay entre log, métrica y traza?
5. ¿Qué mide la distancia de variación total?
6. ¿Qué mide PSI y por qué no se lee sin contexto?
7. ¿Qué es un SLI?
8. ¿Qué es un SLO?
9. ¿Qué significa presupuesto de error?
10. ¿Cómo se calcula el presupuesto consumido de una ventana?
11. ¿Qué significa un burn rate de 5.0 en una ventana de producción?
12. ¿Por qué una ventana como `2026-06-08` queda en `block`?
13. ¿Qué hace `lineage_event.json`?
14. ¿Qué debe contener un runbook útil?
15. ¿Qué slices críticos monitorizarías en tu proyecto?
16. ¿Por qué no deberías reentrenar con una ventana rota?
17. ¿Por qué un retry sin idempotencia puede duplicar efectos?
18. ¿Qué diferencia hay entre backfill y replay?
19. ¿Qué comprobaría un contract test de trazas?
20. ¿Qué acción correctiva añadirías a un postmortem de datos?

21. ¿Qué evidencia pedirías antes de reabrir una ventana bloqueada?
22. ¿Cómo conecta este capítulo con análisis aplicado?
23. ¿Qué herramienta usarías para orquestar, cuál para contratos, cuál para trazas y cuál para drift?
24. ¿Qué evidencia pedirías antes de usar una ventana nueva para reentrenar?
25. ¿Qué SLI usarías para medir freshness en un RAG documental?
26. ¿Por qué `label_delay` puede hacer que una métrica parezca buena demasiado pronto?
27. ¿Qué campos jamás guardarías como labels de Prometheus?
28. ¿Cómo investigarías una incidencia sin guardar prompts crudos?
29. ¿Qué diferencia operativa hay entre replay y backfill?
30. ¿Qué debe congelarse antes de un backfill seguro?
31. ¿Cuándo subirías una alerta de S2 a S1?
32. ¿Qué postmortem te dejaría un test nuevo y no solo una explicación bonita?

En resumen

IDEA	QUÉ TE LLEVAS
DataOps es operación, no decoración.	Contratos, runs, hashes, gates y runbooks sostienen el sistema.
El drift abre investigación.	Un cambio de distribución no se interpreta sin contexto ni consecuencia.
Los SLOs deben ser medibles.	`latency_p95_ms <= 650` es operativo; “que vaya bien” no lo es.
El presupuesto de error se consume.	Si el burn rate se dispara, la ventana se congela hasta tener replay y defensa nueva.
La trazabilidad puede bloquear.	Sin `trace_id`, no hay investigación fiable.
Calidad de datos no es solo drift.	Freshness, completitud, puntualidad y retraso de etiqueta pueden romper una decisión aunque la distribución parezca estable.
La telemetría también tiene privacidad.	Una traza útil debe explicar la operación sin convertirse en almacén de prompts crudos o datos personales.
Los slices siguen vivos en producción.	Lo que se auditó en test debe monitorizarse después.
Un gate protege decisiones.	Evita aumentar automatización o reentrenar con ventanas rotas.
Los retries necesitan idempotencia.	Repetir una tarea no debe duplicar decisiones ni alertas.
Backfill y replay necesitan contrato.	Recalcular histórico sin snapshot, versión y diff aceptado puede destruir evidencia.
La severidad traduce señal en acción.	No todas las alertas pesan igual: impacto, trazabilidad, privacidad y automatización cambian la respuesta.
Un postmortem debe crear una defensa.	La acción correctiva se traduce en test, contrato o gate.

Para saber más

Apache Airflow. (2026). Core Concepts. Documentación

AWS Labs. (2026). Deequ: Unit Tests for Data. GitHub

Baye, C. A. (2016). Emergency Response. En *Site Reliability Engineering*. Google SRE

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. DOI

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. Google Research

Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley.

Dagster. (2026). Dagster documentation. Documentación

dbt Labs. (2026). Model contracts. Documentación

Evidently AI. (2026). Data Drift Documentation. Documentación

European Parliament and Council of the European Union. (2016). Regulation (EU) 2016/679. Eur-Lex

Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En *Site Reliability Engineering*. Google SRE

Great Expectations. (2026). Expectations Overview. Documentación

Jones, C., Wilkes, J., Murphy, N. R. y Beyer, B. (2016). Service Level Objectives. En *Site Reliability Engineering*. Google SRE

Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. Google SRE

National Institute of Standards and Technology. (2020). *NIST Privacy Framework: A Tool for Improving Privacy Through Enterprise Risk Management, Version 1.0*. DOI

OpenLineage. (2026). OpenLineage Documentation. Documentación

OpenTelemetry. (2026). Logs. Documentación

OpenTelemetry. (2026). Metrics. Documentación

OpenTelemetry. (2026). Traces. Documentación

Prefect. (2026). Deployments. Documentación

Prefect. (2026). How to automatically rerun your workflow when it fails. Documentación

Prometheus. (2026). Metric and label naming. Documentación

Sculley, D. y otros (2015). Hidden Technical Debt in Machine Learning Systems. *NeurIPS*. Paper

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

Stribblehill, A. (2016). Managing Incidents. En *Site Reliability Engineering*. Google SRE

Wang, R. Y. y Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33. DOI

Wilkinson, J. (2018). *Alerting on SLOs*. En *The Site Reliability Workbook*. Google SRE

Notas

1. 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 ↩

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

3. 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/ ↩

4. OpenLineage. (2026). *OpenLineage Documentation*. https://openlineage.io/docs/. Consultado el 7 de junio de 2026. ↩

5. OpenTelemetry. (2026). *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/. Consultado el 7 de junio de 2026. ↩

6. OpenTelemetry. (2026). *Metrics*. https://opentelemetry.io/docs/concepts/signals/metrics/. Consultado el 7 de junio de 2026. ↩

7. OpenTelemetry. (2026). *Logs*. https://opentelemetry.io/docs/concepts/signals/logs/. Consultado el 7 de junio de 2026. ↩

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

9. Dagster. (2026). *Dagster documentation*. https://docs.dagster.io/getting-started. Consultado el 7 de junio de 2026. ↩

.0. Prefect. (2026). *Deployments*. https://docs.prefect.io/concepts/deployments. Consultado el 7 de junio de 2026. ↩

.1. Prefect. (2026). *How to automatically rerun your workflow when it fails*. https://docs.prefect.io/v3/how-to-guides/workflows/retries. Consultado el 7 de junio de 2026. ↩

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

.3. Wang, R. Y. y Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33. https://doi.org/10.1080/07421222.1996.11518099 ↩

.4. Great Expectations. (2026). *Expectations Overview*. https://docs.greatexpectations.io/docs/cloud/expectations/expectations_overview/. Consultado el 7 de junio de 2026. ↩

.5. AWS Labs. (2026). *Deequ: Unit Tests for Data*. https://github.com/awslabs/deequ. Consultado el 7 de junio de 2026. ↩

.6. European Parliament and Council of the European Union. (2016). *Regulation (EU) 2016/679*. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679. Consultado el 7 de junio de 2026. ↩

.7. National Institute of Standards and Technology. (2020). *NIST Privacy Framework: A Tool for Improving Privacy Through Enterprise Risk Management, Version 1.0*. https://doi.org/10.6028/NIST.CSWP.01162020 ↩

.8. Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Consultado el 7 de junio de 2026. ↩

.9. 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/ ↩

!0. Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. https://sre.google/sre-book/emergency-response/. Consultado el 7 de junio de 2026. ↩

!1. Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. https://sre.google/sre-book/managing-incidents/. Consultado el 7 de junio de 2026. ↩

!2. 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 ↩

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

24. Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Consultado el 7 de junio de 2026. ↩

25. Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. https://sre.google/sre-book/monitoring-distributed-systems/. Consultado el 7 de junio de 2026. ↩

26. Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. https://sre.google/workbook/alerting-on-slos/. Consultado el 7 de junio de 2026. ↩

27. Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. https://sre.google/sre-book/emergency-response/. Consultado el 7 de junio de 2026. ↩

28. Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. https://sre.google/sre-book/managing-incidents/. Consultado el 7 de junio de 2026. ↩

29. Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. https://sre.google/sre-book/postmortem-culture/. Consultado el 7 de junio de 2026. ↩

Capítulo 07: Análisis aplicado: experimentos, causalidad y decisión

Entrando en el tema

En el capítulo anterior aprendimos a operar datos: ventanas, drift, trazas, SLOs, gates y postmortems. Eso nos protege de tomar decisiones con datos rotos. Ahora damos el siguiente paso: **decidir si una acción cambia algo**.

Esta es una de las fronteras más importantes para cualquier persona que trabaja con IA. Un modelo puede predecir que un estudiante no resolverá su trámite a tiempo. Pero otra pregunta mucho más difícil es: ¿qué acción aumenta realmente la probabilidad de resolverlo? ¿Enviar una plantilla? ¿Escalarlo? ¿Cambiar el orden de cola? ¿Dar un mensaje más claro? ¿Nada?

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Separar predicción de intervención.	No usas un score predictivo como prueba de efecto causal.
Escribir una pregunta causal.	Distingues tratamiento, control, resultado, unidad y población.
Diseñar un experimento mínimo.	Defines asignación, métrica primaria, guardrails, tamaño y criterio de decisión.
Leer un A/B test como artefacto de ingeniería.	Buscas SRM, balance, slices, intervalos y trazabilidad.
Diseñar una plataforma mínima de experimento.	Separas flag, contexto, exposición, métricas, análisis y release gate.
Calcular readiness estadístico.	Escribes MDE, alpha, potencia, muestra y política de peeking.
Entender ATE, CATE y uplift.	Sabes cuándo importa el efecto medio y cuándo importa el efecto por segmento.
Auditar datos observacionales con cuidado.	No cierras una conclusión causal si falta solapamiento o hay confounding.
Generar una decisión reproducible.	Dejas reportes, scorecards y una decisión versionada.

Esta lista no pretende convertirte en estadístico en diez minutos. Pretende darte una brújula de ingeniería: antes de tocar un modelo, un prompt, un RAG o una política automática, debes saber qué decisión vas a permitirte tomar con los datos. Hay experimentos que solo sirven para aprender, otros que sirven para bloquear una release y otros que solo justifican una conversación con más muestra. Confundir esos tres niveles es una fuente silenciosa de malas decisiones.

En este capítulo, cada fórmula y cada artefacto práctico intentan responder a una pregunta operativa. Si una estimación no cambia qué publicas, qué bloqueas, qué repites o qué escalas a revisión humana, todavía no está haciendo trabajo de ingeniería. Por eso aparecen contratos, catálogos de métricas, eventos de exposición y decisiones versionadas junto a causalidad: el número importa, pero el circuito que lo produce importa igual o más.

La frase central:

> Predecir responde qué puede pasar. Causalidad responde qué cambiaría si actuamos.

La escena: el descuento que parecía funcionar

Imagina que tenemos un asistente para soporte académico. El sistema predice qué casos tienen más probabilidad de resolverse tarde. Un equipo propone enviar una plantilla guiada a ciertos casos antes de que el operador responda. Al mirar datos históricos, los casos que recibieron plantilla parecen resolverse mucho más.

La tentación es decir: “perfecto, la plantilla funciona”. Pero quizá la plantilla se mandaba sobre todo a casos de matrícula, que ya tenían más probabilidad de resolverse. O quizá se mandaba a estudiantes con expedientes más completos. O quizá los operadores más expertos usaban más la plantilla y también resolvían mejor sin ella.

El análisis aplicado empieza cuando frenamos esa conclusión rápida y escribimos la pregunta correcta:

PREGUNTA FLOJA	PREGUNTA ÚTIL
¿Quién resuelve mejor?	¿Qué cambia si asignamos la plantilla?
¿Qué variable predice resolución?	¿Qué acción modifica la resolución?
¿La gente con plantilla resolvió más?	¿Casos comparables resolverían más con plantilla que sin plantilla?
¿El modelo lo explica?	¿El diseño permite estimar el efecto?

La diferencia entre esas dos columnas parece lingüística, pero cambia todo el proyecto. En la columna izquierda todavía estamos describiendo el mundo: vemos correlaciones, rankings, grupos que se comportan mejor y variables que anticipan un resultado. En la columna derecha empezamos a diseñar una intervención: definimos qué se cambia, sobre quién, cuándo madura el resultado y qué comparación sería justa.

Esta es una de las razones por las que la causalidad encaja tan bien en ingeniería de IA. Un sistema predictivo puede ordenar una cola y ser útil. Pero en cuanto el sistema actúa, prioriza, recomienda, escala, resume, bloquea o decide qué herramienta usar, ya no basta con predecir. En ese momento necesitas saber si la acción mejora algo o si simplemente estás actuando sobre los casos que ya eran más fáciles.

Qué no es análisis causal

Análisis causal no es mirar una correlación y escribir una historia convincente. Las historias pueden ayudar a formular hipótesis, pero no convierten una asociación en efecto.

Tampoco es una capa que arregle datos malos por sí sola. Si la asignación está rota, si el resultado se mide tarde, si cambió el producto a mitad del experimento o si falta trazabilidad, la fórmula no salva la decisión. Por eso este capítulo depende tanto del capítulo 06 sobre DataOps.

Y no es interpretabilidad. SHAP, importancia de variables, PDP, ICE o trazas pueden ayudarnos a preguntar mejor, pero no prueban por sí solas que una acción cause un resultado. Si una variable aparece como importante, quizá sea causa, quizá sea proxy, quizá sea síntoma o quizá esté recogiendo una regla de negocio.

Qué sí es análisis aplicado para IA

Análisis aplicado es convertir datos en una decisión defendible. En IA suele aparecer en tres momentos:

MOMENTO	PREGUNTA	EJEMPLO
Antes de publicar	¿Este cambio mejora algo real?	Nuevo prompt, nuevo ranking, nueva política de revisión.
Después de publicar	¿La mejora se mantiene?	Ventana post-release comparada con control o holdout.
Al priorizar acciones	¿Dónde conviene intervenir?	Casos donde una plantilla cambia la resolución, no donde ya iba a resolverse.

En equipos reales, estas tres situaciones se mezclan. Un nuevo prompt puede mejorar la tasa de respuesta aceptada, pero quizá empeore citas, latencia o coste. Un nuevo reranker puede recuperar mejores documentos, pero solo para consultas largas. Una regla de revisión humana puede reducir errores graves, pero saturar a un equipo operativo. La causalidad aplicada no busca una medalla para la variante ganadora; busca separar mejora real, coste operativo y riesgo residual.

Rubin formalizó el enfoque de resultados potenciales: para cada unidad imaginamos el resultado bajo tratamiento y bajo control, aunque solo observemos uno.[1] Holland popularizó la idea de que la inferencia causal se enfrenta a un problema fundamental: no podemos observar simultáneamente ambos resultados potenciales para la misma unidad.[2] Pearl desarrolló un marco gráfico y el operador `do` para razonar sobre intervención, ajuste y supuestos causales.[3]

Fecha de corte: **7 de junio de 2026**. Fuentes consultadas ese día: literatura clásica de causalidad, Microsoft ExP, CUPED, DoWhy, EconML, OpenFeature, LaunchDarkly, Statsig, GrowthBook y trabajos sobre calidad de experimentos a escala. Lo estable es la diferencia entre asociación, intervención, contrafactual y decisión. Lo que cambia son herramientas, plataformas y automatizaciones.

Predicción e intervención no responden lo mismo

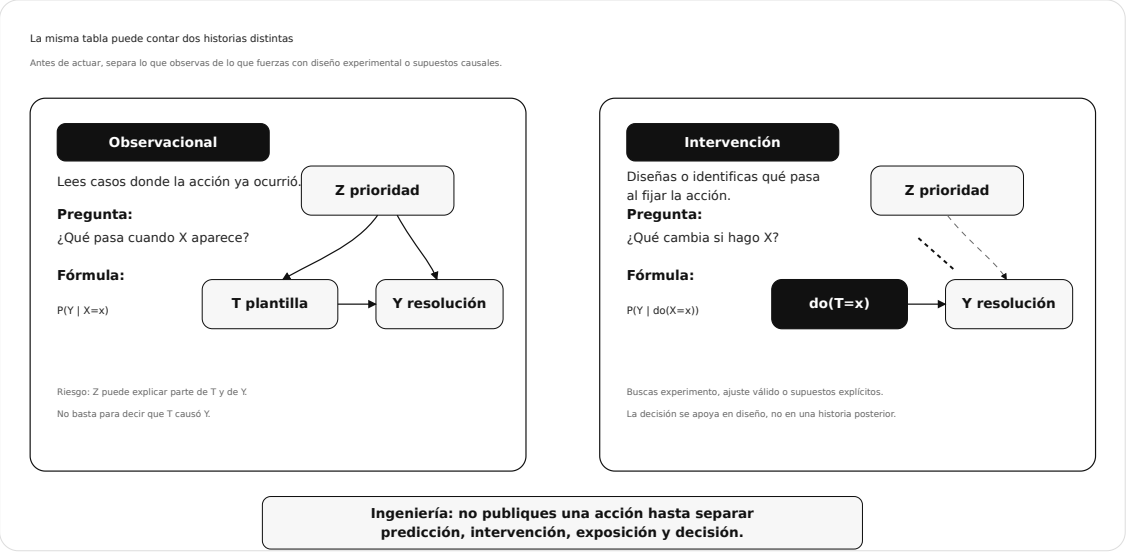
La confusión más habitual cabe en dos expresiones parecidas:

$$P(Y \mid X = x)$$

$$P(Y \mid do(X = x))$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Y	Resultado que medimos.	Caso resuelto.
X	Variable o acción que miramos.	Recibir plantilla.
x	Valor concreto de X.	`plantilla = sí`.
$P(Y \mid X = x)$	Probabilidad observada dado que X ocurrió.	Resolución entre casos que recibieron plantilla.
$do(X = x)$	Intervención que fija X desde el diseño.	Asignar plantilla por experimento.
$P(Y \mid do(X = x))$	Probabilidad bajo intervención.	Resolución si forzamos plantilla en casos comparables.

La primera expresión es predictiva u observacional. Nos dice qué pasa entre casos donde X aparece. La segunda es causal. Nos pregunta qué pasa si intervenimos.



> El operador `do` no es un adorno matemático: marca que la acción se fija por diseño o por un supuesto causal defendible.

Para un ingeniero de IA, la diferencia es práctica:

SI HACES...	NECESITAS...	POR QUÉ
Ordenar casos por riesgo	Predicción bien evaluada.	Quieres anticipar qué ocurrirá.
Cambiar una política	Efecto causal o experimento.	Quieres saber qué cambia por actuar.
Elegir a quién enviar una acción	Uplift o CATE.	No basta saber quién tiene más probabilidad de éxito.
Publicar un cambio de producto	Diseño experimental y guardrails.	Debes proteger métricas secundarias y operación.

El problema de los resultados potenciales

La notación de resultados potenciales escribe dos mundos para cada unidad:

$$Y_i(1)$$

$$Y_i(0)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
i	Unidad analizada.	Ticket `u014`.
$Y_i(1)$	Resultado de la unidad si recibe tratamiento.	Resolvería con plantilla.
$Y_i(0)$	Resultado de la unidad si recibe control.	Resolvería sin plantilla.
1	Tratamiento activo.	`treatment`.
0	Control.	`control`.

El problema es que para una misma unidad solo observamos una de las dos columnas. Si `u014` recibió plantilla, vemos $Y_{014}(1)$, pero no vemos $Y_{014}(0)$. Ese resultado no observado es el contrafactual.

Por eso no medimos el efecto individual real de cada caso. Estimamos efectos agregados bajo supuestos:

$$ATE = E[Y(1) - Y(0)]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
ATE	Efecto medio del tratamiento.	+0.25 en tasa de resolución.
$E[\cdot]$	Esperanza o promedio poblacional.	Media sobre los casos del experimento.
$Y(1)$	Resultado bajo tratamiento.	Resolver con plantilla.
$Y(0)$	Resultado bajo control.	Resolver sin plantilla.

Si aleatorizamos bien, el grupo control actúa como aproximación del mundo sin tratamiento y el grupo tratamiento como aproximación del mundo con tratamiento.

Un A/B test como sistema de ingeniería

Kohavi, Longbotham, Sommerfield y Henne trataron los experimentos online como práctica central para tomar decisiones en la web, pero también documentaron problemas habituales de diseño, métrica y ejecución.[4] Microsoft ExP describe una plataforma de experimentación

con portal, servicio de ejecución, procesamiento de logs y análisis como piezas separadas de una arquitectura escalable.[5]

Esto es importante: un A/B test no es una tabla con `control` y `treatment`. Es un sistema.

PIEZA	PREGUNTA DE INGENIERÍA	FALLO TÍPICO
Unidad	¿Qué se asigna?	Mezclar usuario, sesión y ticket.
Asignación	¿Cómo se decide control o tratamiento?	Reparto roto o no reproducible.
Persistencia	¿La unidad conserva variante?	Un usuario cambia de grupo entre eventos.
Métrica primaria	¿Qué decide el experimento?	Optimizar una proxy cómoda.
Guardrails	¿Qué no puede empeorar?	Mejorar resolución subiendo coste o latencia.
Logging	¿Qué evento demuestra exposición?	Medir casos que nunca vieron la intervención.
Análisis	¿Qué estimador y regla de parada usamos?	Mirar cada día y parar cuando conviene.
Decisión	¿Qué hacemos con el resultado?	Publicar sin contrato ni evidencia.

La tabla se puede leer como una arquitectura mínima. Si falla la unidad, el resto del análisis queda contaminado porque no sabemos qué estamos comparando. Si falla la asignación, no sabemos si la variante llegó a quien debía. Si falla la exposición, contamos unidades que nunca vivieron el tratamiento. Si falla el logging, no podemos reconstruir la historia. Y si falla la decisión, el experimento se convierte en un informe bonito que nadie sabe usar.

Esta mirada es especialmente importante en IA porque el tratamiento suele ser más difuso que en una web clásica. No siempre cambias un botón. A veces cambias el prompt, el modelo, el proveedor, el `temperature`, el chunking, la política de tools, el umbral de revisión o la plantilla de salida. Si no registras esa versión exacta, mañana no podrás repetir el experimento ni explicar por qué una variante funcionó.

Plan de análisis antes de tocar resultados

Un experimento serio se empieza escribiendo antes qué se va a mirar. Esto no es solemnidad académica: es protección contra cambiar la pregunta cuando ya hemos visto la respuesta.

El cuaderno del facsímil incluye `analysis_plan.json`. Ese archivo fija hipótesis, población, tratamiento, control, métrica primaria, guardrails, slices a reportar, exclusiones y reglas de decisión. También incluye `metric_catalog.json`, porque una métrica sin definición estable se convierte en una palabra elástica.

PIEZA DEL PLAN	PREGUNTA QUE RESPONDE	EJEMPLO
Hipótesis	¿Qué creemos que cambiará?	La plantilla guiada aumenta resolución a 7 días.
Población	¿Sobre quién vale la decisión?	Casos académicos con trazabilidad completa.
Unidad	¿Qué se asigna y analiza?	`unit_id`.
Tratamiento	¿Qué cambia exactamente?	Plantilla guiada + contexto documental revisado.
Control	¿Contra qué comparamos?	Flujo actual.
Métrica primaria	¿Qué decide el experimento?	`resolved`.
Ventana	¿Cuándo madura la métrica?	`day_7`.
Guardrails	¿Qué no puede empeorar?	Feedback, latencia, coste, cita válida.
Exclusiones	¿Qué se elimina antes del análisis?	Sin exposure event, fallback no etiquetado, unidad inestable.
Regla de decisión	¿Qué significa `pass`, `review` o `block`?	Publicar solo con readiness y guardrails en `pass`.

Un buen plan de análisis es una defensa contra uno mismo. Cuando ya has visto una gráfica favorable, todo el mundo encuentra razones para cambiar la ventana, mover una exclusión, mirar otro segmento o explicar que la métrica importante era otra. El plan escrito antes reduce esa elasticidad. No elimina el juicio, pero deja claro cuándo estás ejecutando el análisis pactado y cuándo estás explorando.

En una organización, además, el plan de análisis permite repartir responsabilidades. Producto puede defender la hipótesis, ingeniería puede defender la instrumentación, datos puede defender la métrica y operación puede defender los guardrails. Sin esa separación, el experimento acaba siendo una reunión donde cada persona trae una intuición distinta y ninguna se puede auditar.

El detalle importante es “antes”. Si primero miramos el resultado y luego elegimos la métrica que más favorece a la variante, no estamos evaluando; estamos buscando una justificación.

`validate_experiment_design.py` revisa que el plan, catálogo y contrato de flag encajen. No calcula efecto. Hace algo anterior y muy de ingeniería: comprueba si el experimento está suficientemente definido para poder medir.

El catálogo de métricas cumple otra función: evita que dos personas usen la misma palabra para cosas distintas. En un proyecto real, `resolved` no puede significar "el operador cerró el ticket" un día y "el estudiante no volvió a escribir" al siguiente. Si la métrica decide publicación, tiene que tener unidad, ventana, dirección y definición operativa.

CAMPO EN `METRIC_CATALOG.JSON`	QUÉ OBLIGA A DECIDIR	EJEMPLO DEL CUADERNO
`name`	Nombre estable para código, SQL y reporte.	`resolved`.
`type`	Papel de la métrica.	`primary`, `guardrail`, `diagnostic`.
`unit`	Grano de medición.	`unit_id`.
`window`	Cuándo se considera madura.	`day_7` o `exposure`.
`direction`	Qué significa mejorar.	`higher_is_better` o `lower_is_better`.
`definition`	Interpretación que debe sobrevivir al cambio de equipo.	Caso resuelto correctamente dentro de la ventana observada.

Este catálogo parece pequeño, pero evita una cantidad enorme de ambigüedad. En sistemas de IA, una misma palabra puede esconder métricas distintas: "calidad" puede ser aceptación del usuario, evaluación humana, similitud con una respuesta de referencia, precisión de citas, ausencia de toxicidad o reducción de escalados. Si no se escribe la semántica, cada dashboard termina midiendo una cosa distinta con el mismo nombre.

Esto parece burocrático hasta que falla. Si un experimento mejora `answer_accepted` pero empeora `citation_valid`, o si `latency_ms` se mide desde puntos distintos del backend, no tienes una decisión: tienes una conversación interminable. El catálogo corta esa ambigüedad antes de ejecutar.

Plataforma experimental para sistemas de IA

Cuando un equipo de IA madura, el experimento deja de vivir en una hoja de cálculo. Se convierte en una plataforma con piezas separadas. OpenFeature define una especificación abierta para feature flags con una API común y proveedores intercambiables.[6] La

especificación de `evaluation context` describe el contexto que viaja con una evaluación de flag, incluyendo un `targeting key` que identifica la unidad sobre la que se evalúa la variante.[7]

La idea que nos interesa no es la marca de la herramienta. Es el contrato de plataforma:

COMPONENTE	QUÉ HACE	QUÉ DEBE QUEDAR TRAZADO
Feature flag	Decide qué variante recibe una unidad.	`flag_key`, versión, proveedor y regla.
Evaluation context	Aporta atributos para evaluar la flag.	`targeting_key`, segmento, región, canal, entorno.
Assignment service	Hace persistente la variante.	Unidad, variante, timestamp, hash de regla.
Exposure event	Registra que la unidad vio la variante.	Evento de exposición, no solo asignación teórica.
Metrics layer	Define métricas con una semántica estable.	Nombre, ventana, unidad, filtros, owner.
Warehouse	Une exposición, eventos y métricas.	Tablas, particiones, linaje y retrasos de datos.
Analysis service	Calcula efecto, intervalos, slices y guardrails.	Método, parámetros, fecha, versión de contrato.
Release gate	Convierte análisis en acción.	`pass`, `review`, `block`, rollout o rollback.

La separación por capas tiene una ventaja muy concreta: permite depurar. Si el resultado parece extraño, no necesitas discutir todo a la vez. Puedes mirar si la flag asignó bien, si el contexto llevaba el `targeting_key`, si la exposición se emitió, si el warehouse unió tarde los eventos, si la métrica se calculó con la ventana correcta o si el release gate interpretó mal el estado. Sin capas, cada incidente se convierte en una niebla.

En proyectos de IA esto también ayuda a comparar proveedores y modelos sin reescribir todo el sistema. El proveedor de LLM puede cambiar, el framework de agentes puede cambiar y el motor de RAG puede cambiar, pero el contrato experimental debería seguir pidiendo lo mismo: unidad, variante, exposición, métrica, evidencia y decisión.

LaunchDarkly describe experimentación conectando métricas a flags y configuraciones para medir cambios en comportamiento, y menciona A/A testing para validar instrumentación antes de un A/B real.[8] GrowthBook se presenta como plataforma open source de feature flags y experimentación, con énfasis en métricas sobre datos existentes y transparencia de consultas.[9] Statsig documenta opciones avanzadas como duración de asignación, periodo de análisis, covariables, sequential testing y correcciones por múltiples comparaciones.[10]

En la práctica aparecen más opciones: Eppo encaja bien cuando el equipo quiere experimentación apoyada en su propio stack de datos y feature flags de bajo acoplamiento.[11] PostHog combina flags, analítica de producto y experimentos dentro de una misma plataforma, útil cuando un equipo pequeño quiere cerrar el circuito sin montar una plataforma propia desde cero.[12] Optimizely Feature Experimentation se sitúa más cerca de una plataforma enterprise con SDKs, APIs y experimentación en múltiples superficies.[13] Evidently no es una plataforma de causalidad, pero ayuda a vigilar drift, calidad de datos y métricas de sistemas de IA antes o después de un experimento.[14]

En IA, esta arquitectura tiene matices propios:

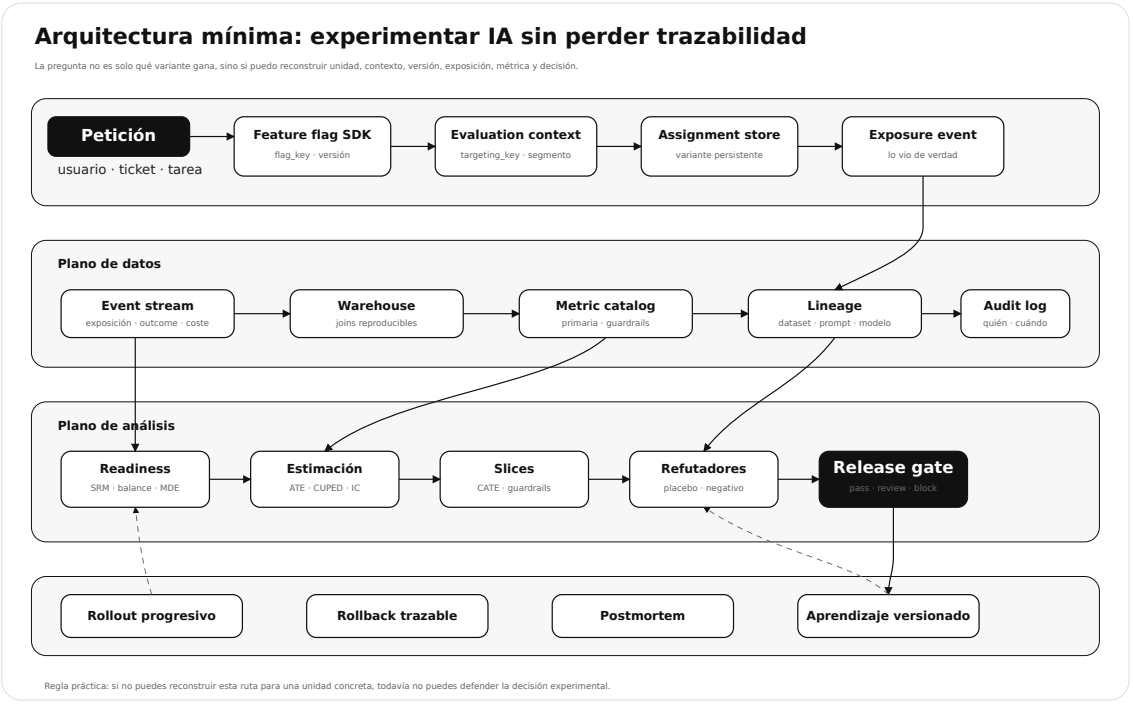
CASO DE IA	QUÉ SE EXPERIMENTA	RIESGO DE INGENIERÍA
Nuevo prompt	Variante textual o plantilla.	La exposición debe registrar versión exacta del prompt.
Nuevo modelo	Modelo, proveedor o configuración.	Latencia, coste y fallback importan tanto como calidad.
RAG	Retriever, reranker, chunking o fuente.	Cambia el contexto recuperado, no solo la respuesta.
Agente	Política de herramientas o aprobación.	La unidad puede ser tarea, conversación o usuario.
Recomendación	Ranking o regla de diversidad.	Un usuario puede afectar inventario, cola o experiencia de otros.
Moderación de cola	Umbral o política de revisión.	Guardrails humanos y operativos son obligatorios.

Fíjate en que “tratamiento” no significa siempre “modelo A contra modelo B”. En un RAG puede ser la forma de partir documentos; en un agente puede ser exigir aprobación humana antes de una herramienta; en un sistema local puede ser una cuantización; en un clasificador puede ser el umbral de derivación. Esta amplitud obliga a documentar la intervención con precisión. Si el tratamiento es ambiguo, el efecto también lo será.

La pregunta de ingeniería no es “¿qué librería uso?”. Es esta:

```
¿Puedo reconstruir qué unidad recibió qué variante,
con qué contexto, bajo qué versión, y qué métricas maduraron después?
```

Si la respuesta es no, todavía no hay experimento publicable.



La plataforma separa ejecución, datos, análisis y decisión. Esa separación evita confundir “la variante se asignó” con “la variante se expuso, maduró y permite decidir”.

Unidad, exposición e interferencia

La unidad de asignación es una decisión de diseño. Si asignas por sesión, pero el usuario vuelve varias veces, puede ver control y tratamiento. Si asignas por ticket, pero el operador aprende de una variante y cambia su comportamiento en otros tickets, hay contaminación entre unidades. Si asignas por empresa, tendrás menos unidades pero menos mezcla entre grupos.

UNIDAD	SIRVE CUANDO	CUIDADO
Usuario	La experiencia se mantiene en el tiempo.	Varias sesiones deben conservar variante.
Sesión	La intervención dura solo una visita.	No sirve para métricas que maduran por usuario.
Ticket	Cada caso es independiente.	Un mismo operador puede influir varios tickets.
Empresa	Hay efecto compartido entre usuarios.	Menos muestra y más varianza.
Conversación	Agentes o asistentes conversacionales.	Una conversación puede contener varias tareas.

La unidad no se elige por comodidad, sino por independencia y por consecuencia. Si el efecto de una variante se queda dentro de una sesión, la sesión puede servir. Si el efecto acompaña a un usuario durante días, una sesión es demasiado pequeña. Si varios usuarios de una misma empresa comparten políticas, documentos o límites, quizá la empresa sea la unidad honesta. La muestra más grande no siempre es la más creíble.

En IA conversacional aparece una trampa adicional: una conversación puede contener varias tareas y cada tarea puede activar herramientas distintas. Si asignas por mensaje, puedes mezclar variantes dentro de la misma conversación. Si asignas por usuario, quizá arrastres aprendizaje o memoria. Si asignas por tarea, necesitas detectar bien dónde empieza y termina. Esta decisión parece técnica, pero cambia qué puedes afirmar al final.

La exposición también importa. Asignar una variante no significa que la unidad la haya recibido. En un sistema de IA puede haber fallback, timeout, caché, error de proveedor, ruta alternativa o una regla de permisos que impide mostrar la variante. Por eso el contrato del cuaderno exige `experiment_exposure`: una decisión experimental se analiza sobre unidades expuestas, no solo asignadas.

Interferencia significa que una unidad puede afectar el resultado de otra. En producto digital aparece en marketplaces, colas, rankings, soporte compartido y sistemas con capacidad limitada. En IA aparece también cuando un agente consume herramientas compartidas, cambia una cola humana, modifica documentos o prioriza casos que compiten por el mismo recurso. Si hay interferencia fuerte, el A/B clásico por usuario puede quedarse corto y quizá necesites asignar por clúster, equipo, cola o ventana temporal.

El cuaderno del facsímil incluye `cluster_interference_events.csv`. La idea es sencilla: si dentro de un mismo equipo conviven control y tratamiento, y el operador aprende de la variante nueva, el control puede contaminarse. En ese caso quizá la unidad correcta no sea `unit_id`, sino `cluster_id`.

SEÑAL	LECTURA
Cluster con control y treatment mezclados	Posible contaminación de comportamiento.
Mismo operador en ambas variantes	Puede aprender de treatment y aplicarlo a control.
Cola compartida	Una variante puede cambiar tiempos de la otra.
Recurso limitado	La mejora de un grupo puede desplazar capacidad.

La interferencia es incómoda porque rompe una suposición que solemos dar por hecha: que cada unidad vive su tratamiento aislada del resto. En una cola de soporte, eso casi nunca es del todo cierto. Si una variante libera tiempo de operadores, puede mejorar también al

control. Si una variante consume más revisión humana, puede empeorar al control. Si un agente aprende una estrategia que un operador imita, la frontera entre grupos se vuelve borrosa.

Una regla práctica:

```
Si las unidades comparten operador, inventario, cola, aula, empresa o herramienta limitada,
pregunta si deberías asignar por clúster.
```

Si asignas por clúster, la pregunta cambia ligeramente. Ya no preguntas solo por tickets individuales; preguntas por equipos, empresas, aulas o colas completas. Donner y Klar desarrollan el diseño y análisis de ensayos aleatorizados por clúster, donde la unidad de asignación y la dependencia dentro de grupo son parte central del análisis.[15] Una forma pedagógica de verlo es calcular primero el resultado medio de cada clúster y después comparar clústeres:

$$\widehat{ATE}_{cluster} = \frac{1}{K_T}\sum_{k \in T} \bar{Y}_k - \frac{1}{K_C}\sum_{k \in C} \bar{Y}_k$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
k	Clúster completo.	`team_a`, `team_b`, `team_c`.
$\bar{Y}_k$	Resultado medio dentro del clúster k.	Resolución media del equipo.
K_T	Número de clústeres en tratamiento.	Equipos asignados a la variante nueva.
K_C	Número de clústeres en control.	Equipos asignados al flujo actual.

La ventaja es que reduces contaminación entre unidades. El precio es que tienes menos observaciones efectivas: cien tickets dentro de tres equipos no equivalen a cien unidades independientes si el equipo comparte operador, cola y aprendizaje. Por eso la decisión de unidad no es un detalle técnico pequeño; cambia la potencia, el coste y la credibilidad del experimento.

Una forma clásica de explicarlo viene del muestreo por clúster. Kish popularizó el *design effect* o efecto de diseño para aproximar cuánto se infla la varianza cuando las observaciones dentro de un clúster se parecen entre sí.[16] Una versión habitual se escribe:

$$DEFF = 1 + (m - 1)\rho$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$DEFF$	Efecto de diseño.	`1.45` .
m	Tamaño medio del clúster.	10 tickets por equipo.
ρ	Correlación intraclúster aproximada.	`0.05` .

En palabras: si los casos dentro de un mismo equipo se parecen, cada ticket aporta menos información independiente de lo que parece. Con $m = 10$ y $\rho = 0.05$, el efecto de diseño es $1 + 9 \cdot 0.05 = 1.45$. Una lectura rápida del tamaño efectivo sería:

$$n_{eff} \approx \frac{n}{DEFF}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
n_{eff}	Tamaño muestral efectivo aproximado.	100 tickets se parecen a unos 69 casos independientes.
n	Número de observaciones registradas.	100 tickets.
$DEFF$	Factor que descuenta dependencia por clúster.	`1.45` .

En palabras: no estás tirando datos; estás dejando de contarlos dos veces. Para un experimento de IA con operadores, aulas, empresas, colas o herramientas compartidas, esta idea evita una trampa muy habitual: diseñar el A/B por evento cuando el comportamiento real se mueve por grupo.

El estimador básico en un A/B test es diferencia de medias:

$$ATE = \bar{Y}_T - \bar{Y}_C$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\widehat{ATE}$	Estimación del efecto medio.	`0.25`.
$\bar{Y}_T$	Media del resultado en tratamiento.	`0.833333` resueltos.
$\bar{Y}_C$	Media del resultado en control.	`0.583333` resueltos.
T	Grupo tratamiento.	Casos con plantilla guiada.
C	Grupo control.	Casos sin plantilla nueva.

En el cuaderno, el efecto observado es:

$$ATE = 0.833333 - 0.583333 = 0.25$$

Esto suena fuerte, pero no basta. Hay que mirar incertidumbre. La forma siguiente es la aproximación habitual del error estándar de una diferencia de medias con grupos independientes, base de los intervalos y tests introductorios que verías en un curso clásico de inferencia estadística.[17]

$$\widehat{SE(ATE)} = \sqrt{\frac{s_T^2}{n_T} + \frac{s_C^2}{n_C}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
SE	Error estándar de la estimación.	`0.186339`.
s_T^2	Varianza del resultado en tratamiento.	Varianza de `resolved` en treatment.
s_C^2	Varianza del resultado en control.	Varianza de `resolved` en control.
n_T	Número de unidades en tratamiento.	`12`.
n_C	Número de unidades en control.	`12`.

Y un intervalo aproximado:

$$IC_{95\%} = ATE \pm 1.96 \cdot SE(ATE)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$IC_{95\%}$	Intervalo de confianza aproximado al 95%.	`[-0.115224, 0.615224]`.
1.96	Multiplicador normal aproximado para 95%.	Valor estándar.
ATE	Efecto estimado.	`0.25`.
$\widehat{SE}$	Error estándar.	`0.186339`.

La lectura honesta es: señal positiva, todavía imprecisa. Publicar automáticamente sería precipitado. Repetir o ampliar muestra es más defendible.

Calidad del experimento antes de mirar el resultado

Antes de celebrar un efecto, un ingeniero revisa si el experimento merece confianza. En el cuaderno lo hacemos con tres controles: SRM, balance y guardrails.

SRM significa *sample ratio mismatch*: el reparto observado no coincide con el reparto esperado. Si diseñamos 50/50 y aparece 70/30, quizá la asignación, tracking o filtrado está fallando. Nie y colaboradores describen validaciones automáticas de aleatorización y detección de SRM en plataformas de experimentación a escala.[18]

Una comprobación clásica usa chi-cuadrado:

$$\chi^2 = \sum_{g \in G} \frac{(O_g - E_g)^2}{E_g}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
χ^2	Estadístico de desajuste.	`0.0`.
G	Grupos del experimento.	`control`, `treatment`.
O_g	Conteo observado en grupo g.	`12`.
E_g	Conteo esperado en grupo g.	`12`.

El balance revisa si las covariables previas al tratamiento son parecidas. Si los grupos ya eran distintos antes de actuar, el efecto puede mezclarse con composición.

Una medida simple es la diferencia de medias estandarizada. En estudios observacionales y emparejamiento con propensity score se usa mucho como diagnóstico de balance, porque expresa diferencias de covariables en unidades comparables.[19]

$$SMD = \frac{\bar{X}_T - \bar{X}_C}{s_{pooled}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
SMD	Diferencia estandarizada entre grupos.	`0.0`.
$\bar{X}_T$	Media de covariable previa en tratamiento.	Tareas previas en treatment.
$\bar{X}_C$	Media de covariable previa en control.	Tareas previas en control.
s_{pooled}	Desviación típica combinada.	Variación compartida entre grupos.

Los guardrails son métricas que no deciden la victoria, pero sí pueden impedir publicar. En el cuaderno, la intervención mejora resolución, pero vigilamos coste, latencia y feedback negativo.

Esta parte es donde muchos experimentos “ganadores” dejan de serlo. Una variante que resuelve más tickets puede ser peor si duplica coste, degrada latencia, aumenta errores de cita o desplaza trabajo a revisión humana. En IA, los guardrails no son decoración ética ni prudencia abstracta: son métricas de supervivencia del sistema. Si el usuario recibe mejores respuestas pero el equipo no puede pagar el coste o auditar las citas, el experimento no ha ganado de forma completa.

Tamaño muestral, MDE y peeking

El error más común al enseñar A/B testing es quedarse en la fórmula del efecto y no hablar de cuánta muestra hace falta para medirlo. Un experimento con 12 unidades por variante puede ser excelente para aprender el mecanismo, pero no para tomar una decisión global si queremos detectar un cambio pequeño.

El MDE (*minimum detectable effect*) es el efecto mínimo que el experimento está diseñado para detectar con una potencia determinada. La fórmula siguiente es la aproximación normal clásica para tamaño muestral en dos grupos de igual tamaño; no es una ocurrencia del capítulo, sino una pieza estándar de cálculo de potencia y tamaño muestral.[20]

$$n \approx \frac{2\left(z_{1-\alpha/2} + z_{1-\beta}\right)^2 \sigma^2}{\delta^2}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
n	Unidades necesarias por variante.	`1530` para el MDE del cuaderno.
α	Probabilidad aceptada de falso positivo.	`0.05`.
$1-\beta$	Potencia: probabilidad de detectar el efecto si existe.	`0.8`.
$z_{1-\alpha/2}$	Cuantil normal para el nivel de confianza.	`1.959964`.
$z_{1-\beta}$	Cuantil normal para la potencia.	`0.841621`.
σ^2	Varianza aproximada de la métrica.	Para binaria: $p(1-p)$.
δ	MDE que queremos detectar.	`0.05`.

Para una métrica binaria como `resolved`, usamos:

$$\sigma^2 = p(1-p)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p	Tasa base esperada.	`0.58`.
$1-p$	Complemento de la tasa base.	`0.42`.
σ^2	Varianza binaria aproximada.	`0.2436`.

El cuaderno calcula:

PARÁMETRO	VALOR
Baseline	`0.58`
Alpha	`0.05`
Potencia	`0.8`
MDE planificado	`0.05`
n actual por variante	`12`
n recomendado por variante	`1530`
MDE aproximado con n actual	`0.564504`

Esta tabla es una de las más importantes del capítulo porque traduce una intuición a una restricción dura. Con pocas unidades puedes detectar solo efectos enormes. Eso no invalida el experimento: puede servir para validar instrumentación, confirmar que el pipeline produce artefactos y aprender si los guardrails se mueven. Pero no deberíamos venderlo como prueba definitiva de impacto global.

Para un equipo de ingeniería, el MDE cambia la conversación. En vez de preguntar “¿ha salido significativo?”, preguntas “¿qué cambio mínimo necesitábamos detectar y cuánta muestra hacía falta para eso?”. Si el negocio necesita detectar una mejora de dos puntos y tu experimento solo puede detectar veinte, el resultado negativo no demuestra ausencia de efecto; demuestra que el diseño no tenía suficiente resolución.

Eso explica por qué el capítulo deja el experimento en `review`: la señal observada es positiva, pero la muestra solo permite detectar efectos enormes. Para decidir con un MDE pequeño, necesitamos más unidades.

Peeking es mirar resultados muchas veces y decidir cuando la gráfica parece favorable. Statsig recomienda configurar sequential testing cuando se quiere evitar que mirar varias veces aumente falsos positivos.[21] En un contrato serio, la regla debe estar escrita antes:

DECISIÓN	QUÉ ESCRIBIR ANTES DE EMPEZAR
Cuándo mirar	Fecha final, número de looks o regla secuencial.
Qué métrica decide	Una primaria, no cinco primarias cambiantes.
Qué guardrails bloquean	Umbrales concretos y dirección.
Qué pasa si queda `review`	Ampliar muestra, repetir ventana o limitar rollout.
Qué no se permite	Cambiar metodología al ver el resultado.

Johari, Pekelis y Walsh explican por qué mirar repetidamente un A/B test y parar cuando conviene aumenta el riesgo de falso positivo si el análisis no lo contempla.[22] La solución profesional no es “no mires nunca”, porque producto y operación necesitan seguimiento. La solución es declarar una regla de mirada.

ESTRATEGIA	QUÉ HACE	CUÁNDO ENCAJA	RIESGO SI SE EXPLICA MAL
Mirada única	Analizas al final de la ventana.	Experimentos simples y métricas maduras.	Tardas en detectar problemas operativos.
Sequential testing	Permite mirar varias veces con control del error.	Producto necesita vigilancia durante la prueba.	Parece “barra libre” si no se fija antes.
Alpha spending	Reparte el presupuesto de error entre miradas.	Hay calendario de revisiones planificado.	La implementación debe ser la misma que el análisis.
Límites Pocock	Usa umbral parecido en varias miradas.	Quieres regla sencilla de comunicación.	Puede ser menos agresivo al principio.
Límites O'Brien-Fleming	Umbral muy estricto al principio y más flexible al final.	Quieres parar pronto solo con evidencia muy fuerte.	Es más difícil de explicar a negocio.

Pocock y O'Brien-Fleming son diseños clásicos de análisis secuencial en ensayos, pero la lección nos sirve aquí: si vas a mirar varias veces, esa política forma parte del experimento, no de la conversación posterior.[23][24]

También conviene ejecutar A/A antes de A/B. En A/A, ambas variantes se comportan igual. Si el sistema detecta efecto donde no debería, tenemos problema de instrumentación, asignación, logging o análisis. Es una prueba humilde y muy de ingeniería: antes de medir impacto, comprobamos que el contador mide.

Métricas, jerarquía y comparaciones múltiples

Otro tropiezo habitual: medir muchas cosas y quedarse con la que sale bonita. Si miras diez métricas, cinco slices y tres ventanas, alguna diferencia llamativa aparecerá por azar. Por eso el plan debe separar métrica primaria, guardrails y diagnósticas.

TIPO	DECIDE	EJEMPLO	CÓMO SE INTERPRETA
Primaria	Sí	`resolved_day_7` .	Una, fijada antes.
Guardrail	Bloquea	`latency_ms` , `cost_eur` , `citation_valid` .	No gana el experimento, pero puede impedir publicar.
Diagnóstica	Explica	`retrieval_precision` .	Ayuda a depurar, no decide sola.
Slice	Matiza	`segment=becas` .	Señal de heterogeneidad o riesgo.
Ventana secundaria	Complementa	`day_1` , `day_30` .	Ayuda a maduración, no reemplaza primaria sin plan.

La jerarquía de métricas es una forma de honestidad. Una primaria decide porque representa la apuesta principal. Los guardrails protegen lo que no quieres romper. Las diagnósticas explican mecanismos y ayudan a depurar. Los slices muestran dónde la media puede estar escondiendo daño o beneficio. Si todo decide, nada decide; si nada decide, el experimento no cambia la acción.

Si declaramos muchos resultados como “primarios”, sube el riesgo de falso descubrimiento. Una corrección conservadora es Bonferroni:

$$\alpha^* = \frac{\alpha}{m}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
α^*	Umbral corregido por número de pruebas.	`0.01` si $\alpha = 0.05$ y $m = 5$.
α	Umbral original.	`0.05` .
m	Número de pruebas consideradas.	`5` guardrails o hipótesis.

Benjamini y Hochberg propusieron controlar la tasa de descubrimientos falsos, menos conservadora que Bonferroni cuando se exploran muchas hipótesis.[25]

En este facsímil la regla pedagógica es clara:

```
Una métrica primaria.
Guardrails bloqueantes.
Slices y diagnósticas como explicación, no como celebración automática.
```

CUPED: reducir varianza sin cambiar la pregunta

CUPED usa información previa al experimento para reducir ruido. Deng, Xu, Kohavi y Walker propusieron usar datos pre-experimento para mejorar la sensibilidad de experimentos online.[26] Microsoft ExP lo describe como técnica de reducción de varianza en A/B testing.[27]

La idea no es manipular el resultado. Es restar parte del ruido explicable por una covariable medida antes del tratamiento:

$$Y_i^* = Y_i - \theta(X_i - \bar{X})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Y_i^*	Resultado ajustado por CUPED.	`resolved` ajustado.
Y_i	Resultado original.	`resolved = 1`.
X_i	Covariable previa al experimento.	`historical_resolution_rate`.
$\bar{X}$	Media de la covariable previa.	Media histórica del dataset.
θ	Peso del ajuste.	`3.107096` en el cuaderno.

El peso suele estimarse así:

$$\theta = \frac{Cov(Y, X)}{Var(X)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$Cov(Y, X)$	Covarianza entre resultado y covariable previa.	Relación entre resolución y histórico.
$Var(X)$	Varianza de la covariable previa.	Dispersión de `historical_resolution_rate`.
θ	Coeficiente de ajuste.	`3.107096`.

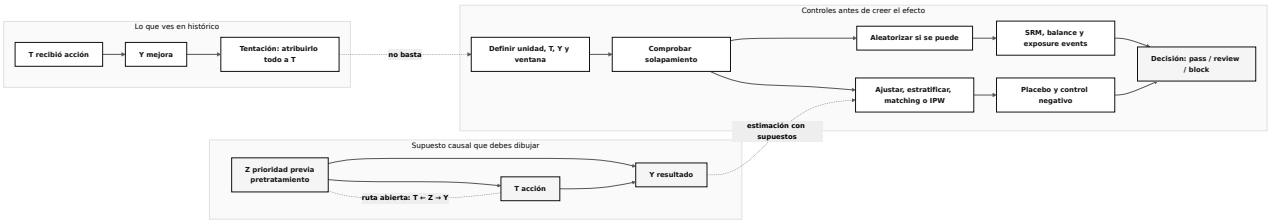
En el cuaderno, CUPED mantiene el efecto en `0.25`, pero baja el error estándar de `0.186339` a `0.162557`. Eso no convierte una señal en verdad absoluta, pero ayuda a medir con menos ruido.

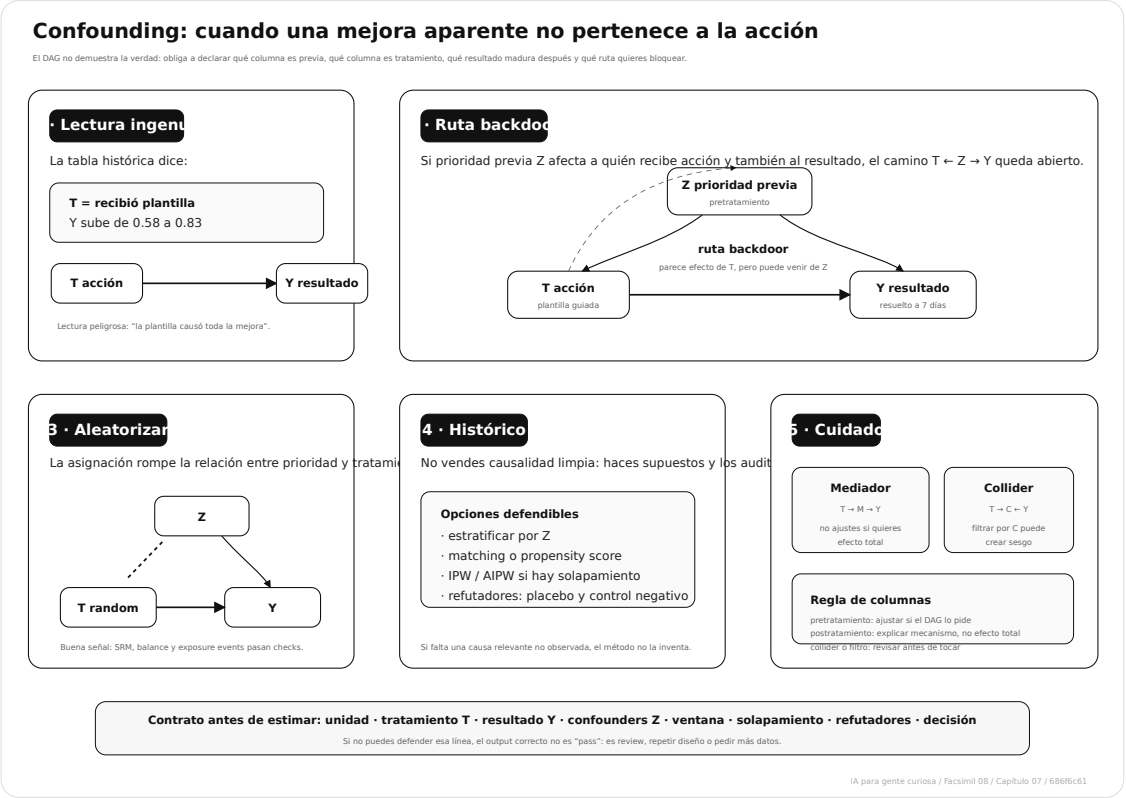
La intuición es parecida a escuchar una conversación con ruido de fondo y conocer de antemano parte de ese ruido. Si antes del experimento ya sabemos que algunos casos eran históricamente más fáciles, podemos usar esa información previa para reducir variabilidad. Pero hay una condición clave: la covariable debe estar medida antes del tratamiento. Si usas una variable afectada por la intervención, ya no estás reduciendo ruido; estás cambiando la pregunta.

Causalidad observacional: cuando no puedes aleatorizar

No siempre podemos hacer un A/B test. A veces trabajamos con datos históricos. En ese caso, el primer deber es no vender la asociación como efecto.

El problema típico se llama confounding. Una variable Z influye en el tratamiento y en el resultado:





> El confounding no es "una variable más": es una ruta alternativa que puede explicar por qué tratamiento y resultado se mueven juntos. Primero se dibuja el supuesto; luego se elige experimento, ajuste o revisión.

El dibujo tiene una consecuencia práctica muy seria: antes de estimar nada hay que decidir qué columnas pertenecen al pasado, cuáles describen la acción y cuáles son resultado. En un sistema de IA esto se vuelve fácil de ensuciar. Una prioridad calculada después de que el modelo intervenga no puede usarse como si fuese contexto previo. Una etiqueta humana generada tras ver la respuesta del asistente no es una variable inocente. Un score de riesgo que cambia durante la conversación puede ser parte del mecanismo, no una causa previa que podamos ajustar sin pensar.

Este punto suele separar un análisis defendible de un informe vistoso. Si el equipo no puede explicar por qué Z ocurre antes que T, por qué T ocurre antes que Y, y por qué la ventana de Y es suficiente para que el efecto madure, el análisis causal todavía no está listo. Dibujar el DAG no garantiza que tengas razón, pero obliga a decir dónde podría estar el error. Esa humildad técnica es exactamente lo que se necesita cuando un número va a decidir si una plantilla, un modelo o un flujo de revisión se publica.

Si no ajustamos por Z, podemos atribuir a T lo que en realidad viene de la prioridad previa. El ajuste por backdoor escribe:

$$P(Y \mid do(T = t)) = \sum_{z} P(Y \mid T = t, Z = z) P(Z = z)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T	Tratamiento o acción.	Recibir plantilla.
t	Valor del tratamiento.	`1` o `0`.
Y	Resultado.	Caso resuelto.
Z	Confounder observado.	Prioridad previa.
z	Nivel concreto de Z.	`high`, `medium`, `low`.
$P(Y \mid T = t, Z = z)$	Resultado dentro de un estrato comparable.	Resolución de casos `medium` con plantilla.
$P(Z = z)$	Peso del estrato en la población.	Proporción de prioridad `medium`.

Esto exige solapamiento. Si en prioridad `low` nadie recibió tratamiento, no podemos comparar tratamiento contra control dentro de ese nivel. El cuaderno lo muestra: el efecto ingenuo es `0.5`, pero al estratificar por prioridad baja a `0.0375` en la población estimable y queda en `review` por falta de solapamiento.

DoWhy separa modelado causal, identificación, estimación y refutación.[28] Su documentación actual insiste en tratar los supuestos como ciudadanos de primera clase y separar identificación de estimación.[29] EconML aplica técnicas de machine learning a estimación de respuestas causales individuales o heterogéneas en datos experimentales u observacionales.[30]

La regla para el facsímil es sencilla: si no puedes escribir supuestos, estimando, población y prueba de robustez, todavía no tienes conclusión causal.

Una herramienta clásica en observacional es el propensity score: la probabilidad de recibir tratamiento dado el contexto observado.[31] Se escribe:

$$e(x) = P(T = 1 \mid X = x)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$e(x)$	Propensity score.	Probabilidad de recibir plantilla.
$T = 1$	Recibir tratamiento.	`received_action = 1`.
$X = x$	Contexto observado.	Prioridad, segmento, histórico.

Sirve para emparejar, ponderar o estratificar casos con probabilidades parecidas de recibir tratamiento. Pero no resuelve el sesgo por sí solo: solo ajusta por variables observadas. Si falta una causa importante, el sesgo puede seguir ahí. Imbens y Rubin desarrollan con detalle este tipo de inferencia causal basada en supuestos explícitos.[32]

Métodos observacionales que conviene reconocer

En ingeniería de IA vas a ver datasets históricos donde nadie aleatorizó nada: campañas, prompts usados por operadores, colas priorizadas, descuentos, plantillas, cambios de ranking, modelos servidos solo a ciertos clientes o herramientas activadas por segmento. En esos casos no hay magia. Hay familias de métodos, cada una con supuestos y fallos.

MÉTODO	QUÉ INTENTA HACER	SIRVE CUANDO	NO SIRVE SI
Matching	Comparar casos tratados con controles parecidos.	Tienes covariables observadas ricas y buen solapamiento.	Hay variables importantes no medidas o no hay controles comparables.
IPW	Ponderar por la probabilidad inversa de recibir tratamiento.	Quieres reconstruir una población más comparable con propensity score.	Hay propensity cercano a 0 o 1; los pesos explotan.
Estimador doblemente robusto	Combinar modelo de resultado y modelo de tratamiento.	Puedes modelar resultado y asignación con cierto cuidado.	Ambos modelos están mal o faltan variables clave.
Difference-in-differences	Comparar cambios antes/después entre tratado y control.	Hay una intervención temporal y tendencia paralela plausible.	Los grupos ya venían cambiando distinto antes de la intervención.
Regression discontinuity	Mirar unidades cerca de un umbral de asignación.	La regla de asignación tiene corte claro.	La gente manipula el umbral o no hay suficientes casos cerca.
Variable instrumental	Usar una variable que empuja el tratamiento sin afectar directamente al resultado.	El instrumento es defendible y fuerte.	El instrumento afecta al resultado por otro camino.

Estos métodos no son una escalera donde uno sea “mejor” que el anterior. Son respuestas distintas a problemas distintos. Matching intenta construir comparaciones parecidas. IPW intenta reponderar una población que no fue asignada al azar. Difference-in-differences aprovecha cambios temporales. Regression discontinuity explota una regla de corte. Variables instrumentales buscan una fuente externa de variación. El error de ingeniería es elegir el método por moda o por disponibilidad de librería, no por el mecanismo que produjo los datos.

Para un alumno de datos, la pregunta útil es: ¿qué historia tendría que ser cierta para que este método fuese razonable? Si digo matching, necesito creer que mis covariables observadas capturan las diferencias relevantes. Si digo difference-in-differences, necesito mirar tendencias previas. Si digo instrumento, necesito defender que el instrumento no afecta al resultado por otro camino. Esta parte no se resuelve con más código; se resuelve con linaje, conocimiento de negocio y una discusión honesta de supuestos.

Una formulación clásica de ponderación por probabilidad inversa, emparentada con el estimador de Horvitz-Thompson, estima el efecto medio así:[33]

$$\widehat{ATE}_{IPW} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{T_i Y_i}{e(X_i)} - \frac{(1-T_i)Y_i}{1-e(X_i)}\right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\widehat{ATE}_{IPW}$	Efecto medio estimado con ponderación inversa.	Efecto de enviar plantilla ajustando por probabilidad de recibirla.
T_i	Indicador de tratamiento.	`1` si recibió plantilla.
Y_i	Resultado observado.	`resolved`.
$e(X_i)$	Propensity score.	Probabilidad estimada de recibir plantilla dado el contexto.
n	Número de unidades.	Casos históricos observados.

En palabras: si un caso tenía poca probabilidad de recibir tratamiento y aun así lo recibió, pesa más porque representa situaciones raras dentro del histórico. El riesgo es que pesos muy grandes conviertan unas pocas filas en dueñas de la conclusión. Por eso el solapamiento no es un detalle: es una condición de trabajo.

Los estimadores doblemente robustos combinan un modelo de resultado $\hat{\mu}_t(X)$ con un propensity score $\hat{e}(X)$. Bang y Robins los desarrollan en el contexto de datos faltantes e inferencia causal.[34] Una forma habitual del estimador AIPW es:

$$\hat{\tau}_{AIPW} = \frac{1}{n}\sum_{i=1}^{n}\left[\hat{\mu}_1(X_i) - \hat{\mu}_0(X_i) + \frac{T_i(Y_i - \hat{\mu}_1(X_i))}{\hat{e}(X_i)} - \frac{(1-T_i)(Y_i - \hat{\mu}_0(X_i))}{1-\hat{e}(X_i)}\right]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{\tau}_{AIPW}$	Efecto estimado con ajuste doblemente robusto.	Impacto de una política sobre resolución.
$\hat{\mu}_1(X_i)$	Resultado esperado si el caso i recibe tratamiento.	Resolución esperada con plantilla.
$\hat{\mu}_0(X_i)$	Resultado esperado si el caso i queda en control.	Resolución esperada sin plantilla.
$\hat{e}(X_i)$	Propensity score estimado.	Probabilidad de recibir plantilla.
T_i, Y_i	Tratamiento y resultado observados.	Acción y resolución en el dataset.

En palabras: el estimador intenta corregir dos cosas a la vez: cómo se asignó el tratamiento y qué resultado esperaba cada tipo de caso. No autoriza a apagar el criterio. Si los datos no tienen las variables que explican la asignación real, el estimador puede seguir siendo elegante y equivocado.

Cuando existe un antes y un después, *difference-in-differences* compara cambios, no niveles. Card y Krueger lo usaron en un estudio clásico sobre salario mínimo y empleo.[35] La forma básica es:

$$\hat{\delta}_{DID} = (\bar{Y}_{T,post} - \bar{Y}_{T,pre}) - (\bar{Y}_{C,post} - \bar{Y}_{C,pre})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{\delta}_{DID}$	Estimación difference-in-differences.	Cambio atribuible a publicar una nueva política.
$\bar{Y}_{T,post}$, $\bar{Y}_{T,pre}$	Resultado medio del grupo tratado después y antes.	Resolución tras activar plantilla y antes.
$\bar{Y}_{C,post}$, $\bar{Y}_{C,pre}$	Resultado medio del grupo control después y antes.	Resolución en grupo sin plantilla.

En palabras: si ambos grupos venían evolucionando igual antes del cambio, la diferencia de cambios puede aproximar el efecto. Si las tendencias previas ya eran distintas, el método puede convertir una tendencia natural en causalidad falsa.

Regression discontinuity se usa cuando una regla asigna tratamiento al cruzar un umbral: por ejemplo, “si el score de riesgo supera 0.80, entra revisión humana”. Lee y Lemieux resumen este diseño en econometría aplicada.[36] La lectura ingenieril es comparar casos muy cerca del

corte: un caso con 0.799 y otro con 0.801 deberían parecerse mucho salvo por la regla. Si el score se manipula, si el umbral cambia sin versionado o si hay pocos casos cerca, la conclusión se debilita.

Las variables instrumentales entran cuando hay una variable Z que cambia la probabilidad de tratamiento, pero no debería afectar al resultado salvo a través de ese tratamiento. Angrist, Imbens y Rubin formalizaron la identificación de efectos causales con instrumentos.[37] En el caso simple, el estimador de Wald se escribe:

$$\hat{\beta}_{IV} = \frac{Cov(Z, Y)}{Cov(Z, T)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{\beta}_{IV}$	Efecto estimado mediante instrumento.	Impacto de recibir revisión humana inducida por una regla externa.
Z	Instrumento.	Disponibilidad aleatoria de un equipo revisor.
T	Tratamiento.	Recibir revisión humana.
Y	Resultado.	Resolver correctamente.
$Cov(\cdot, \cdot)$	Covarianza entre variables.	Asociación del instrumento con tratamiento y resultado.

En palabras: el instrumento tiene que mover el tratamiento, pero no puede tener otro camino hacia el resultado. Es una condición fuerte. En un sistema de IA, muchas variables que parecen instrumentos no lo son: turno, canal, país o proveedor suelen afectar otras cosas además del tratamiento.

Refutar antes de creerse el número

DoWhy llama *refuters* a pruebas que intentan cuestionar una estimación: añadir una causa común aleatoria, usar un tratamiento placebo, mirar subconjuntos o probar sensibilidad.[38] El cuaderno añade una versión mínima y ejecutable de esa idea. No pretende competir con DoWhy, EconML o CausalML; pretende que el gesto quede claro.

REFUTADOR DEL CUADERNO	QUÉ PREGUNTA	QUÉ HARÍA SALTAR UNA ALERTA
Tratamiento placebo desplazado	¿Aparece efecto incluso con una acción falsa?	Un efecto placebo grande.
Variable previa como control negativo	¿La acción ya estaba asociada a algo anterior?	Diferencias fuertes en `historical_resolution_rate`.
Estabilidad por segmento	¿El efecto cambia demasiado por slice?	Un promedio global que esconde heterogeneidad.
Solapamiento por prioridad	¿Hay tratamiento y control en cada estrato?	Estratos con solo acción o solo control.

Los refutadores son una forma sana de trabajar contra nuestras ganas de tener razón. En un proyecto real siempre hay presión para convertir una señal prometedora en lanzamiento, sobre todo si la variante coincide con lo que el equipo quería probar. Un placebo, un control negativo o una revisión por segmento no son trabas académicas: son ensayos de resistencia. Si una conclusión causal es frágil, conviene descubrirlo antes de meterla en una política de producto, una cola de soporte o una decisión automatizada.

El mensaje para ingeniería es sobrio: una estimación observacional debe sobrevivir a intentos razonables de refutación antes de convertirse en decisión. Si se rompe al primer placebo, no es una derrota del equipo; es una victoria del sistema de revisión.

Uplift y CATE: actuar donde cambia algo

El ATE puede ser positivo y aun así esconder una decisión mala. Quizá la intervención no ayuda a todos por igual. En IA aplicada, muchas veces queremos saber dónde cambia algo.

El efecto condicionado se escribe:

$$CATE(x) = E[Y(1) - Y(0) \mid X = x]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$CATE(x)$	Efecto medio condicionado al contexto x.	Efecto en `segment=becas`.
$X = x$	Contexto o características del caso.	Segmento, idioma, canal, prioridad.
$Y(1)$	Resultado con tratamiento.	Resolución con plantilla.
$Y(0)$	Resultado sin tratamiento.	Resolución sin plantilla.

Esto conecta directamente con slices. En el cuaderno:

SEGMENTO	CONTROL	TRATAMIENTO	EFECTO OBSERVADO
`becas`	`0.25`	`0.75`	`0.50`
`matricula`	`1.00`	`1.00`	`0.00`
`practicas`	`0.50`	`0.75`	`0.25`

Esta lectura cambia el tipo de producto que construirías. Si el efecto vive sobre todo en `becas`, quizá no necesitas encarecer todo el flujo con una plantilla guiada, un modelo más grande o una revisión humana universal. Quizá necesitas una política específica para ese segmento, una evaluación adicional en ese slice y una explicación de por qué ahí sí aparece margen de mejora. CATE y uplift sirven para pasar de “la media sube” a “esta acción merece aplicarse aquí, y no necesariamente allí”.

La decisión no debería ser “plantilla para todo el mundo”. Podría ser: ampliar muestra, estudiar por qué `becas` concentra mejora y comprobar que `matricula` no necesita gasto adicional.

Del experimento al rollout

Un resultado experimental no es el final del trabajo. Es la entrada a una decisión de release. En sistemas de IA, publicar al 100% sin escalones puede mezclar tres problemas a la vez: efecto causal, estabilidad operativa y coste.

Una política de rollout sobria puede ser:

PASO	QUÉ OCURRE	QUÉ SE MIRA
A/A	Variantes iguales.	SRM, exposición, métricas estables.
A/B pequeño	Primera ventana controlada.	Señal, guardrails, slices.
Ramp 5%	Tráfico real limitado.	Latencia, coste, errores y trazas.
Ramp 25%	Más diversidad de casos.	CATE, segmentos raros, operación humana.
Ramp 50%	Estrés realista.	SLOs y presupuesto de error.
100%	Publicación completa.	Monitorización post-release y rollback.

Un rollout no es un premio por haber ganado una tabla. Es otra fase del experimento, con otra clase de riesgos. En tráfico pequeño sueles ver casos limpios; al crecer aparecen idiomas raros, documentos incompletos, picos de latencia, límites de proveedor, operadores saturados y segmentos que no estaban representados. Por eso el rollout tiene que conservar la misma disciplina del experimento: versión, exposición, métricas, guardrails y decisión escrita.

En el contrato del cuaderno aparece:

```
{
  "initial_ramp_percent": 5,
  "ramp_steps_percent": [5, 25, 50, 100],
  "rollback_if_guardrail_blocks": true,
  "publish_requires_status": "pass"
}
```

Esto no es decoración. Si el experimento queda en `review`, no debería pasar a 100%. Puede pasar a otra ventana de medición, a un A/A pendiente, a un rollout limitado o a una mejora de instrumentación. La salida profesional no es “ganó B”. La salida profesional es: **qué hacemos ahora, con qué riesgo y qué evidencia queda guardada**.

Bandits y experimentos adaptativos

Un A/B test clásico mantiene proporciones estables mientras mide. Eso es útil porque simplifica la interpretación: si control y tratamiento están bien asignados, estimar el efecto es directo. Pero a veces el objetivo no es solo medir; también queremos aprender y asignar más tráfico a variantes que parecen mejores durante el proceso.

Ahí aparecen los bandits. En un problema de bandits elegimos acciones, observamos recompensa y vamos actualizando la política de asignación. Sutton y Barto lo explican como una tensión entre exploración y explotación: probar para aprender frente a usar lo que parece mejor.[39]

La idea mínima se puede escribir así:

$$a_t = \arg\max_a Q_t(a)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a_t	Acción elegida en el paso t.	Variante de prompt para el ticket actual.
a	Acción candidata.	`control`, `treatment_a`, `treatment_b`.
$Q_t(a)$	Valor estimado de la acción en el paso t.	Tasa media de resolución observada.
$\arg\max$	Acción con mayor valor estimado.	Variante que parece mejor.

Pero si siempre elegimos la que parece mejor, podemos dejar de explorar demasiado pronto. Una regla epsilon-greedy añade exploración:

$$a_t = \begin{cases} \textbf{acción aleatoria} & \textbf{con probabilidad } \epsilon \\ \arg\max_a Q_t(a) & \textbf{con} \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
ϵ	Probabilidad de explorar.	`0.1`.
$1 - \epsilon$	Probabilidad de explotar lo mejor conocido.	`0.9`.
$Q_t(a)$	Estimación acumulada por acción.	Calidad media por variante.

Para IA aplicada, la tabla de decisión sería:

DISEÑO	ÚTIL CUANDO	NO LO USES SI
A/B fijo	Necesitas una conclusión clara y auditable.	La pérdida de oportunidad durante la prueba es muy alta.
A/A	Quieres validar instrumentación.	Esperas medir efecto real.
Bandit	Quieres aprender y reasignar tráfico mientras operas.	Necesitas estimación simple del ATE con reparto estable.
Rollout progresivo	Quieres publicar con control operativo.	Lo confundes con experimento causal completo.
Holdout persistente	Quieres medir impacto largo plazo.	No puedes mantener grupo sin tratamiento.

La decisión entre A/B, bandit, rollout y holdout depende de qué te duele más: ignorancia, coste de oportunidad, riesgo operativo o pérdida de una referencia estable. Si estás probando un prompt de bajo riesgo y el volumen es alto, quizá un A/B fijo te da una lectura limpia. Si una variante mala puede dañar a usuarios o consumir mucho coste, un rollout progresivo tiene más sentido. Si el sistema aprende mientras sirve, un bandit puede reducir exposición a malas variantes, pero a cambio exige más cuidado al analizar. No hay una opción profesional por defecto; hay una opción coherente con el daño posible.

Los bandits pueden ser muy útiles en recomendación, prompts alternativos o selección de respuesta, pero complican inferencia, métricas tardías y comunicación. Si cambias la asignación durante el experimento, el análisis debe saberlo. No puedes tratarlo luego como si hubiese sido un A/B fijo.

De quién viene cada fórmula y por qué aparece aquí

Este capítulo usa fórmulas porque el tema las necesita. No están para decorar ni para dar una apariencia científica. Cada una corresponde a una familia reconocible de la literatura de inferencia causal, experimentación online, estadística aplicada o aprendizaje por refuerzo. La pregunta editorial es siempre la misma: ¿esta fórmula ayuda a tomar una decisión de ingeniería más limpia?

FÓRMULA O FAMILIA	PROCEDENCIA	QUÉ APORTA AL CAPÍTULO	DECISIÓN QUE AYUDA A TOMAR
$P(Y \mid X = x)$ frente a $P(Y \mid do(X = x))$	Pearl y el marco de intervención con operador `do`.	Separa asociación observada de intervención diseñada.	No usar una correlación histórica como prueba de impacto.
$Y_i(1)$, $Y_i(0)$ y $ATE = E[Y(1) - Y(0)]$	Resultados potenciales de Rubin y formulación causal de Holland.	Explica el problema contrafactual: solo vemos uno de los dos mundos.	Diseñar control y tratamiento antes de hablar de efecto.
$\widehat{ATE} = \bar{Y}_T - \bar{Y}_C$	Estimación clásica de diferencia de medias en experimentos aleatorizados; se apoya en el marco de resultados potenciales de Rubin/Holland cuando la asignación es aleatoria.	Convierte la comparación de variantes en una magnitud interpretable.	Saber si el tratamiento mueve la métrica primaria y cuánto.
$SE(\widehat{ATE})$ e $IC_{95\%}$	Inferencia estadística clásica para diferencia de medias, como se enseña en textos de inferencia estadística.	Añade incertidumbre al efecto observado.	Evitar publicar por una señal grande pero imprecisa.
$\widehat{ATE}_{cluster}$	Ensayos y diseños con aleatorización por clúster, tratados de forma específica por Donner y Klar.	Recuerda que no todas las unidades son independientes.	Cambiar la unidad de asignación cuando hay operador, cola o equipo compartido.
$DEFF = 1 + (m - 1)\rho$ y $n_{eff} \approx n / DEFF$	Muestreo por clúster de Kish.	Traduce dependencia dentro de clúster a tamaño efectivo.	No contar eventos correlacionados como si fueran unidades independientes.
$\chi^2 = \sum_g (O_g - E_g)^2 / E_g$	Test chi-cuadrado usado para detectar desajustes de reparto, incluyendo SRM.	Comprueba si el reparto observado coincide con el esperado.	No analizar un experimento si la aleatorización o el tracking parecen rotos.
$SMD = (\bar{X}_T - \bar{X}_C) / s_{pooled}$	Diferencia de medias estandarizada usada como diagnóstico de balance; Austin la discute en el contexto de propensity score.	Permite comparar grupos antes del tratamiento en una escala común.	Detectar si control y tratamiento ya eran distintos antes de actuar.

FÓRMULA O FAMILIA	PROCEDENCIA	QUÉ APORTA AL CAPÍTULO	DECISIÓN QUE AYUDA A TOMAR
Fórmula aproximada de n para MDE	Cálculo clásico de tamaño muestral y potencia para dos grupos con aproximación normal.	Conecta muestra disponible con efecto detectable.	Distinguir “sirve para aprender” de “sirve para publicar”.
$\alpha^* = \alpha/m$	Corrección de Bonferroni para comparaciones múltiples.	Enseña el coste de mirar muchas métricas como si todas fueran primarias.	Separar métrica primaria, guardrails y diagnósticas.
$Y_i^* = Y_i - \theta(X_i - \bar{X})$ y $\theta = Cov(Y, X)/Var(X)$	CUPED de Deng, Xu, Kohavi y Walker.	Reduce varianza usando una covariable previa al experimento.	Ganar sensibilidad sin cambiar la pregunta causal.
Ajuste por backdoor	Pearl y grafos causales.	Muestra cómo ajustar por un confounder observado cuando no hay experimento.	No vender datos observacionales como causalidad si faltan supuestos o solapamiento.
$e(x) = P(T = 1 \mid X = x)$	Propensity score de Rosenbaum y Rubin.	Resume la probabilidad de recibir tratamiento dado el contexto observado.	Emparejar, ponderar o estratificar sin olvidar que solo ajusta variables observadas.
$\widehat{ATE}_{IPW}$	Ponderación por probabilidad inversa basada en Horvitz-Thompson y propensity score.	Repondera casos según lo raro que era observar su tratamiento.	Ver si el efecto depende de pesos inestables o falta de solapamiento.
$\hat{\tau}_{AIPW}$	Estimación doblemente robusta de Bang y Robins.	Combina modelo de resultado y modelo de asignación.	No depender de una sola familia de modelo cuando se trabaja con histórico.
$\hat{\delta}_{DID}$	Difference-in-differences usado en cuasi-experimentos.	Compara cambios antes/después entre tratado y control.	Evaluar cambios temporales cuando la aleatorización no existe.
$\hat{\beta}_{IV} = Cov(Z, Y)/Cov(Z, T)$	Variables instrumentales de Angrist, Imbens y Rubin en el caso simple.	Usa una fuente externa de	No aceptar un supuesto instrumental débil

FÓRMULA O FAMILIA	PROCEDENCIA	QUÉ APORTA AL CAPÍTULO	DECISIÓN QUE AYUDA A TOMAR
		variación en el tratamiento.	sin discutir exclusión y fuerza.
$CATE(x) = E[Y(1) - Y(0) \mid X = x]$	Efectos heterogéneos de tratamiento en inferencia causal moderna.	Lleva el análisis desde el promedio hacia segmentos accionables.	Decidir si publicar para todos o solo para un slice con evidencia.
$a_t = \arg\max_a Q_t(a)$ y epsilon-greedy	Bandits en aprendizaje por refuerzo, como en Sutton y Barto.	Explica exploración frente a explotación.	Elegir entre A/B fijo, bandit o rollout progresivo según la necesidad de medir o actuar.

La lectura práctica es esta: si una fórmula no cambia qué se diseña, qué se mide, qué se bloquea o qué se publica, sobra. Si cambia una decisión, entonces debe venir con fuente, símbolos, ejemplo y una traducción a operación.

Herramientas y decisiones de compra o montaje

No hace falta que cada equipo construya Microsoft ExP desde cero. Pero sí debe saber qué está comprando o montando.

OPCIÓN	ENCAJA CUANDO	PREGUNTAS ANTES DE USARLA
Feature flags simples	Solo necesitas activar o desactivar variantes.	¿Registra exposición? ¿Persistencia? ¿Auditoría?
Plataforma de experimentación	Quieres análisis, métricas y decisiones recurrentes.	¿SRM, A/A, CUPED, sequential testing, slices?
Warehouse-native	Ya tienes métricas en BigQuery, Snowflake, Databricks o similar.	¿La definición SQL de métricas es versionable?
Implementación propia	Necesitas control completo o restricciones internas.	¿Quién mantiene asignación, logs, análisis y UI?
OpenFeature + proveedor	Quieres no acoplar código a una herramienta.	¿El proveedor soporta contexto, tracking y trazas?

La herramienta correcta depende de qué pieza te falta. Si el problema es que nadie sabe qué variante recibió cada usuario, necesitas mejorar asignación y exposición, no comprar un dashboard bonito. Si el problema es que las métricas viven en el warehouse y cambian cada semana, necesitas contratos de métricas, linaje y revisión SQL. Si el problema es que producto lanza sin evidencia, necesitas release gates. Y si el problema es que el equipo no distingue asociación de causalidad, la plataforma no lo arregla sola.

Lo que no conviene comprar es “confianza”. La confianza se construye con eventos completos, métricas versionadas, checks de SRM, balance, ventanas de maduración, guardrails y revisiones. Una plataforma puede ayudarte a ejecutar esa disciplina con menos fricción, pero no puede decidir por ti si el tratamiento estaba bien definido o si el resultado medido era el que importaba.

En causalidad observacional, la elección de herramienta también importa. DoWhy te fuerza a separar modelo causal, identificación, estimación y refutación. EconML se centra más en estimar efectos heterogéneos con aprendizaje automático. CausalML ofrece métodos de uplift y CATE desde una interfaz Python orientada a casos experimentales u observacionales.[40] PyMC puede servir cuando quieres modelar incertidumbre de forma bayesiana y trabajar con propensity scores o modelos causales probabilísticos.[41]

HERRAMIENTA	MEJOR USO EN ESTE CAPÍTULO	QUÉ NO DEBES DELEGARLE
DoWhy	Escribir DAG, identificar estimando, estimar y ejecutar refutadores.	La validez de tus supuestos causales.
EconML	CATE, uplift y tratamiento heterogéneo con modelos de ML.	La definición de población, tratamiento y resultado.
CausalML	Uplift modeling, meta-learners, árboles uplift y validación práctica.	La auditoría de solapamiento y sesgo no medido.
PyMC	Causalidad con incertidumbre bayesiana y modelos probabilísticos explícitos.	Convertir un mal DAG en una buena conclusión.
GrowthBook, Statsig, LaunchDarkly	Experimentos online, flags, métricas y decisiones de producto.	La semántica de tus métricas o tus guardrails.
Eppo	Feature flags y experimentación conectada al stack de datos existente.	Que tu warehouse tenga métricas bien definidas y contratos de exposición.
PostHog	Experimentos, analítica de producto y flags en equipos que quieren empezar rápido.	Diseñar una pregunta causal limpia o una política de peeking.
Optimizely Feature Experimentation	Experimentación con SDKs y APIs en entornos más enterprise o multi-superficie.	La calidad de tus eventos, tus métricas y tu decisión de rollout.
Evidently	Monitorizar drift, calidad y comportamiento de datos/modelos alrededor del experimento.	Estimar causalidad o sustituir un A/B test.
OpenFeature	Desacoplar código de proveedor de flags.	El análisis estadístico posterior.

Leída así, la tabla no es una lista de logos. Es un mapa de responsabilidades. DoWhy o EconML viven más cerca de la inferencia causal. GrowthBook, Statsig, LaunchDarkly, Eppo, PostHog y Optimizely viven más cerca de flags, producto y análisis recurrente. Evidently ayuda alrededor del ciclo, cuando quieres saber si los datos o el comportamiento del modelo se están moviendo. OpenFeature reduce acoplamiento en código. En un equipo serio estas piezas pueden convivir, pero cada una debe tener un papel claro.

Para IA, añade estas preguntas:

PREGUNTA	POR QUÉ IMPORTA
¿La variante incluye versión de modelo, prompt y parámetros?	Cambiar `temperature`, prompt o modelo cambia el tratamiento.
¿La exposición registra fallback?	Si el modelo nuevo falla y se usa otro, no puedes contar esa unidad igual.
¿Las métricas maduran tarde?	Una resolución puede medirse días después de la exposición.
¿Hay trazas de herramienta o RAG?	Sin ellas no sabes qué contexto o tool produjo el resultado.
¿El coste se mide por unidad asignada o expuesta?	Una variante puede resolver más y aun así no compensar.

Experimentos RAG y métricas tardías

Un experimento RAG no mide solo “la respuesta gustó”. Cambiar el retriever, el reranker o el chunking cambia qué evidencia llega al modelo. Por eso las métricas deben separar resultado, calidad de recuperación y guardrails técnicos.

En el cuaderno añadimos `rag_experiment_events.csv` con estas señales:

MÉTRICA	QUÉ MIDE	POR QUÉ IMPORTA
`answer_accepted`	Si la respuesta fue aceptada.	Métrica de utilidad.
`citation_valid`	Si la cita apuntaba a evidencia válida.	Guardrail documental.
`retrieval_precision`	Si los documentos recuperados eran relevantes.	Calidad del sistema RAG, no solo del LLM.
`latency_ms`	Tiempo de respuesta.	Un reranker puede mejorar calidad y empeorar experiencia.
`cost_eur`	Coste por unidad.	La mejora puede no compensar si el coste sube demasiado.

Un experimento RAG suele fallar cuando mezcla niveles. Si cambias el chunking, estás tocando recuperación; si cambias el prompt, estás tocando generación; si cambias el reranker, estás tocando orden de evidencia; si cambias el modelo, estás tocando lectura y

síntesis. Si todo cambia a la vez y solo miras aceptación final, quizá ganes, pero no sabrás qué aprendiste. Para ingeniería, aprender qué componente movió la métrica importa casi tanto como ganar el experimento.

La salida del cuaderno deja el experimento RAG en `review`: el tratamiento mejora aceptación y recuperación, pero sube latencia y coste. Esa es una decisión realista. En IA aplicada, muchas mejoras son tradeoffs, no victorias limpias.

También añadimos `late_metric_events.csv`. Algunas métricas maduran tarde. Un caso puede parecer resuelto en el día 1 y reabrirse en el día 7. O puede necesitar seguimiento al principio y terminar bien después. Por eso la tabla de métricas debe declarar ventana:

VENTANA	QUÉ PUEDE MEDIR	RIESGO
`day_0`	Exposición, latencia, coste.	Todavía no sabes si ayudó.
`day_1`	Resolución temprana.	Puede ignorar reaperturas.
`day_7`	Resolución más estable.	Llega tarde para decisiones rápidas.
`day_30`	Retención o satisfacción persistente.	Mezcla más cambios del entorno.

La ventana es parte del contrato, no un parámetro menor. En sistemas de soporte, una respuesta puede parecer buena en el momento y generar una reapertura dos días después. En sistemas de recomendación, una acción puede subir clics inmediatos y dañar satisfacción posterior. En un asistente con herramientas, una ejecución rápida puede ocultar una corrección manual posterior. Si la métrica madura tarde, el sistema de decisión tiene que esperar o declarar que decide con una métrica temprana y limitada.

Un sistema profesional no pregunta solo “¿qué métrica?”. Pregunta “¿en qué ventana, con qué unidad y con qué maduración?”.

Schema, CI y contrato de análisis

El cuaderno incluye `warehouse_schema.sql` para dejar una idea de arquitectura mínima. Hay cuatro tablas:

TABLA	QUÉ GUARDA	POR QUÉ EXISTE
`experiment_units`	Unidad, variante, flag y contexto.	Saber qué se asignó.
`exposure_events`	Exposición real a la variante.	Saber qué se vio o recibió.
`metric_events`	Métricas por unidad y ventana.	Separar exposición de resultado.
`experiment_decisions`	Decisión final y evidencia.	Versionar la salida profesional.

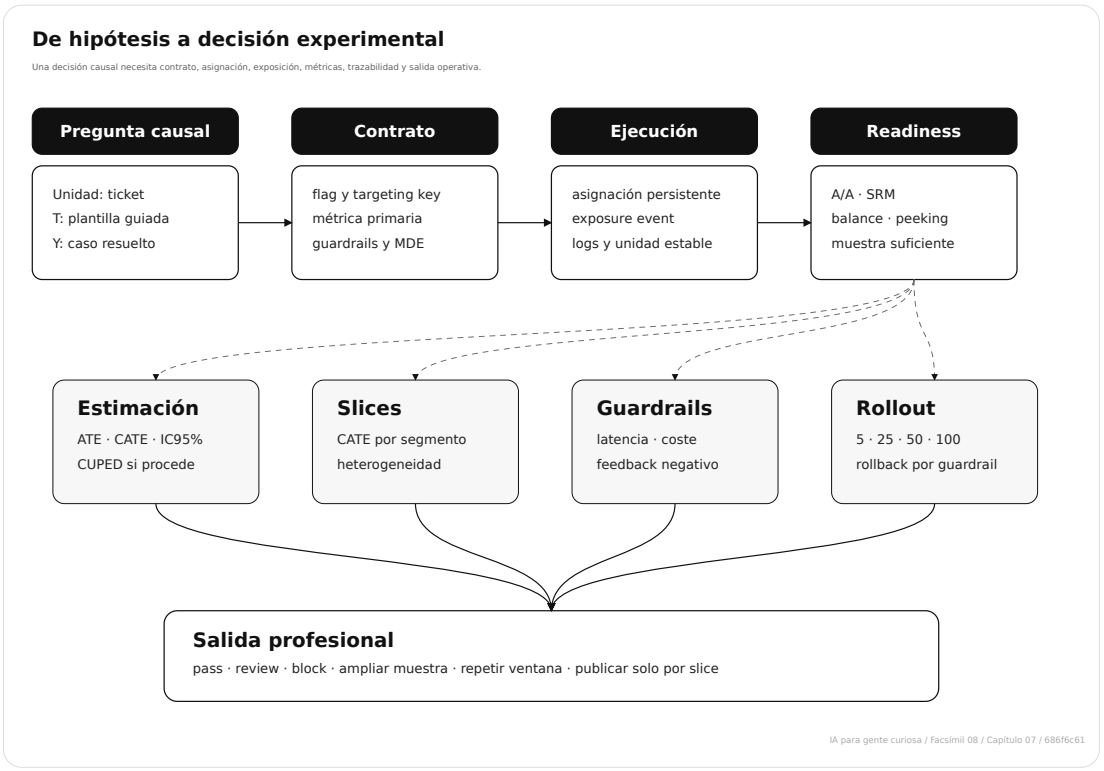
Este schema no es una propuesta universal de base de datos. Es una forma mínima de obligar a que el experimento deje rastro. La unidad asignada, la exposición real y la métrica madura no deberían vivir mezcladas en una misma fila improvisada. Separarlas evita errores muy comunes: contar unidades no expuestas, medir antes de tiempo, perder la versión de la flag o no poder reconstruir por qué una release pasó.

También permite que el CI revise algo más que “el script corre”. Puede comprobar que el análisis produjo un reporte, que los campos críticos existen, que la decisión está escrita y que los guardrails no se han saltado. Esta es la diferencia entre un notebook útil para explorar y un sistema de ingeniería capaz de sostener decisiones repetidas.

Además, `ci_experiment_gate.py` revisa que existan contratos, outputs y campos obligatorios del reporte. No bloquea por estar en `review`; lo deja explícito. Bloquearía si faltan archivos, si el schema falla, si SRM queda en `block` o si algún guardrail bloquea.

Eso es importante para equipos de ingeniería: una decisión en `review` puede ser aceptable si significa “seguir midiendo”. Una decisión en `block` significa “no publiques, hay una condición rota”. El CI no debe sustituir criterio, pero sí debe impedir que un experimento incompleto parezca listo.

Arquitectura de una decisión experimental



> Una decisión experimental no empieza en el p-value: empieza en la pregunta causal y termina en una salida operativa versionada.

En el día a día

En producción no basta con decir que el tratamiento “gana”. El resultado tiene que pasar por un contrato.

En el cuaderno del facsímil, el A/B test tiene 24 unidades: 12 en control y 12 en tratamiento. La métrica primaria es `resolved`, que debe subir. Los guardrails vigilan feedback negativo, latencia y coste.

SEÑAL	RESULTADO	LECTURA
Control `resolved`	`0.583333`	Línea base.
Treatment `resolved`	`0.833333`	Mejora observada.
ATE observado	`0.25`	Señal positiva.
IC95%	`[-0.115224, 0.615224]`	Intervalo demasiado ancho.
SRM	`pass`	Asignación 50/50 correcta.
Balance previo	`pass`	Covariables iniciales equilibradas.
Guardrails	`pass`	Coste y latencia no bloquean.
Decisión	`review`	Prometedor, pero falta precisión.

El readiness añade otra capa:

SEÑAL DE READINESS	RESULTADO	LECTURA
A/A previo	`review`	Falta documentarlo antes de confiar en el A/B.
Exposure event	`pass`	El contrato exige `experiment_exposure`.
Asignación persistente	`pass`	La unidad conserva variante.
n recomendado por variante	`1530`	El MDE de `0.05` exige mucha más muestra.
MDE aproximado con n actual	`0.564504`	Con 12 por variante solo detectas efectos enormes.
Política de peeking	`pass`	Hay una sola mirada planificada.
Rollout	`pass`	Hay pasos 5/25/50/100 y rollback por guardrail.

El análisis observacional cuenta otra historia:

LECTURA	RESULTADO
Efecto ingenuo	`0.5`
Efecto estratificado por prioridad	`0.0375`
Población estimable por prioridad	`0.75`
Decisión	`review`

La diferencia enseña la lección del capítulo: lo que parece enorme en datos históricos puede encogerse cuando comparas contextos más parecidos.

Por qué debería importarte

Si no separas predicción de intervención, puedes construir sistemas que gastan recursos donde no cambian nada. Puedes enviar acciones a quienes ya iban a resolver. Puedes castigar un segmento porque aparece asociado a un resultado, cuando en realidad recibe casos más difíciles. Puedes publicar un cambio por una métrica primaria bonita mientras se degrada latencia, coste o experiencia.

Para ingeniería de IA, causalidad no es lujo académico. Es control de daños en decisiones que actúan sobre el mundo.

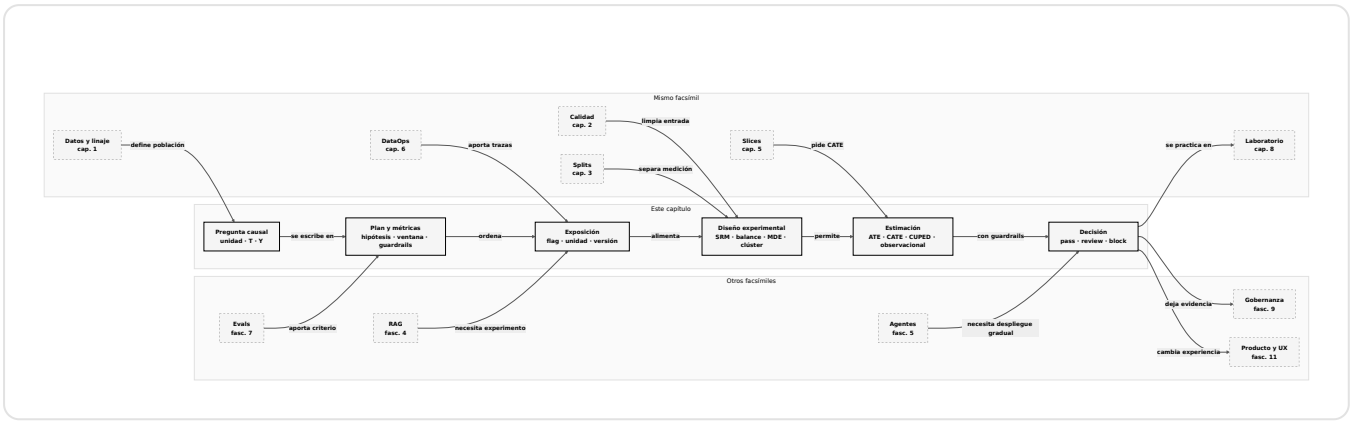
Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Usar predicción como efecto	El score parece accionable.	Preguntar si queremos $P(Y \mid X)$ o $P(Y \mid do(X))$.
Celebrar un ATE sin guardrails	La métrica primaria sube.	Revisar coste, latencia, feedback y slices antes de decidir.
Mirar el resultado antes de validar asignación	Queremos saber quién ganó.	Comprobar SRM, balance y exposición antes del efecto.
Vender datos históricos como experimento	Hay muchas filas y parece serio.	Escribir DAG, confounders, solapamiento y supuestos.
Ignorar heterogeneidad	El promedio es cómodo.	Revisar CATE o slices antes de publicar una política global.
Parar cuando el resultado gusta	La señal temprana seduce.	Fijar criterio de parada y decisión antes de mirar.
No registrar exposición real	La asignación parece suficiente.	Medir solo unidades que vieron o recibieron la variante.
No calcular MDE	El resultado parece grande.	Escribir MDE, alpha, potencia y n antes de empezar.
Pasar de experimento a 100%	El equipo quiere cerrar rápido.	Usar rollout por pasos y rollback si falla un guardrail.
Confundir métrica primaria con métrica diagnóstica	Todas parecen interesantes.	Una primaria decide; las demás explican o bloquean.
Ignorar clústeres naturales	La unidad individual da más muestra aparente.	Preguntar si equipo, empresa, aula, operador o cola comparten efectos.
Ajustar observacional sin mirar solapamiento	El modelo devuelve un número.	Revisar si existen casos comparables antes de hablar de efecto.

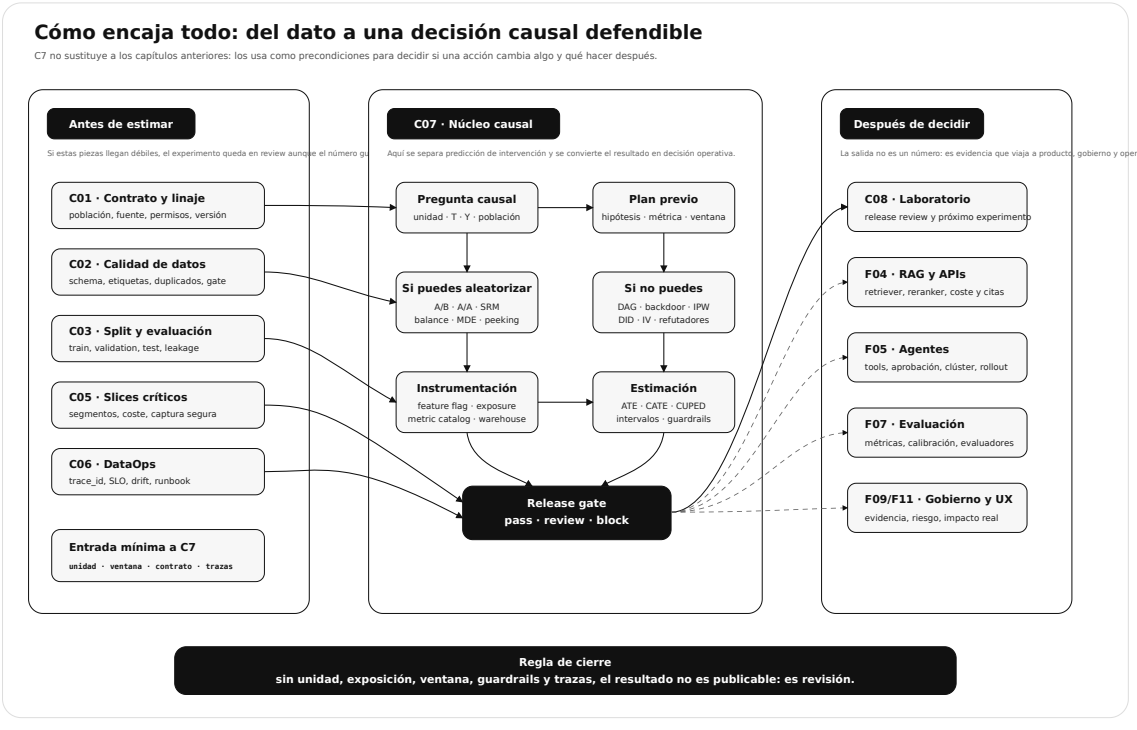
Cómo encaja todo

Este mapa se lee desde los capítulos anteriores. Los datos y contratos vienen del capítulo 01, la evaluación honesta del capítulo 03, los slices del capítulo 05 y la operación del capítulo 06. Este capítulo enseña a decidir si una acción cambia algo.

La decisión que introduce es nueva: no basta con medir calidad predictiva; necesitamos estimar efecto, revisar diseño experimental y dejar una salida operativa. En el cierre del facsímil, esto se convertirá en cierre integrador.



El Mermaid anterior sirve para estudiar relaciones. El SVG siguiente lo baja a arquitectura de decisión: qué piezas deben llegar sanas desde datos, qué decide este capítulo y qué salidas deja para laboratorio, gobernanza y producto. Es el dibujo que conviene tener en la cabeza cuando alguien dice “la variante ganó” demasiado rápido.



C7 encaja como la capa de decisión causal: recibe datos auditables, define intervención y exposición, estima efecto con incertidumbre y entrega una decisión que el laboratorio, la gobernanza y el producto pueden defender.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Tratamiento	Acción o variante cuyo efecto queremos medir.
Control	Condición de comparación sin la intervención nueva.
Resultado	Variable que medimos después de la intervención.
Unidad	Elemento que se asigna y analiza: usuario, ticket, empresa o sesión.
ATE	Efecto medio del tratamiento.
CATE	Efecto medio condicionado a un contexto o segmento.
Contrafactual	Resultado que no observamos para la misma unidad bajo otra acción.
Confounder	Variable que afecta al tratamiento y al resultado.
DAG causal	Grafo que explicita hipótesis sobre relaciones causales.
SRM	Desajuste entre asignación esperada y observada.
Guardrail	Métrica que no debe empeorar aunque la primaria mejore.
CUPED	Ajuste con covariable previa para reducir varianza.
Solapamiento	Existencia de tratamiento y control dentro de contextos comparables.
Feature flag	Interruptor controlado en ejecución para servir variantes sin desplegar código nuevo.
Evaluation context	Contexto usado para evaluar una flag: unidad, segmento, canal, entorno o atributos.
Exposure event	Evento que demuestra que la unidad recibió la variante.
A/A test	Prueba con variantes equivalentes para validar instrumentación y reparto.
MDE	Efecto mínimo detectable bajo una muestra y potencia dadas.
Peeking	Mirar resultados repetidamente sin regla previa.
Rollout	Publicación progresiva con pasos, guardrails y rollback.
Plan de análisis	Contrato previo con hipótesis, población, métrica, ventanas, exclusiones y reglas de decisión.

TÉRMINO	DEFINICIÓN BREVE
Catálogo de métricas	Registro versionado que define nombre, unidad, ventana, dirección y propósito de cada métrica.
Comparaciones múltiples	Riesgo de encontrar señales aparentes al mirar muchas métricas, slices o ventanas.
Propensity score	Probabilidad estimada de recibir tratamiento dado el contexto observado.
Asignación por clúster	Diseño que asigna grupos completos cuando las unidades pueden influirse entre sí.
Efecto de diseño	Factor que aproxima cuánto aumenta la varianza cuando las observaciones dentro de un clúster se parecen entre sí.
Refutador causal	Prueba que intenta romper una conclusión causal con placebo, control negativo, subconjuntos o supuestos alternativos.
Control negativo	Variable o prueba que no debería cambiar por la intervención y sirve para detectar sesgo o mala instrumentación.
IPW	Ponderación por probabilidad inversa para ajustar por la probabilidad estimada de recibir tratamiento.
Estimador doblemente robusto	Estimador que combina modelo de resultado y propensity score para reducir dependencia de un único modelo.
Difference-in-differences	Diseño cuasi-experimental que compara cambios antes/después entre grupo tratado y grupo control.
Regression discontinuity	Diseño que estima efecto cerca de un umbral de asignación.
Variable instrumental	Variable que afecta al tratamiento y solo debería afectar al resultado a través de ese tratamiento.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Qué diferencia hay entre $P(Y \mid X)$ y $P(Y \mid do(X))$?
2. ¿Por qué no observamos $Y_i(1)$ y $Y_i(0)$ a la vez?
3. ¿Qué mide el ATE?
4. ¿Qué mide el CATE?
5. ¿Qué es un contrafactual?

6. ¿Qué piezas debe tener una pregunta causal?
7. ¿Qué comprueba SRM?
8. ¿Qué significa balance previo entre grupos?
9. ¿Por qué un guardrail puede bloquear un experimento ganador?
10. ¿Qué hace CUPED y qué no hace?
11. ¿Qué es un confounder?
12. ¿Por qué hace falta solapamiento en datos observacionales?
13. ¿Por qué un experimento puede quedar en `review` en lugar de `pass`?
14. ¿Por qué el efecto ingenuo observacional no basta?
15. ¿Qué demuestra un exposure event?
16. ¿Por qué un A/A test puede ahorrar un experimento mal medido?
17. ¿Qué significa MDE?
18. ¿Por qué mirar resultados muchas veces sin regla previa cambia el riesgo de error?
19. ¿Qué debería contener una política de rollout?
20. ¿Qué campos mínimos debe tener un plan de análisis?
21. ¿Qué diferencia hay entre métrica primaria, guardrail y diagnóstica?
22. ¿Cuándo usarías Bonferroni o control de descubrimientos falsos?
23. ¿Qué intenta resumir un propensity score?
24. ¿Por qué una asignación por clúster puede ser más honesta aunque tenga menos potencia?
25. ¿Qué significa efecto de diseño y por qué reduce tamaño efectivo?
26. ¿Qué diferencia hay entre confounder, mediador y collider?
27. ¿Qué intenta detectar un tratamiento placebo?
28. ¿Cuándo usarías IPW y cuándo desconfiarías de sus pesos?
29. ¿Por qué un estimador doblemente robusto no elimina la necesidad de buenos supuestos?
30. ¿Qué supuesto central necesita difference-in-differences?
31. ¿Qué condición fuerte necesita una variable instrumental?
32. ¿Cómo conecta este capítulo con el cierre del facsímil?

En resumen

IDEA	QUÉ TE LLEVAS
Predicción no es intervención.	Un score útil para anticipar no prueba que una acción cambie el resultado.
Un experimento es arquitectura.	Unidad, asignación, logging, métricas y análisis deben estar contratados.
El efecto necesita incertidumbre.	Un ATE observado sin intervalo puede llevar a decisiones precipitadas.
Los guardrails importan.	Una mejora primaria no justifica degradar coste, latencia o experiencia.
Causalidad observacional exige supuestos.	Si falta solapamiento o hay confounding, la lectura queda en triage.
Las decisiones se versionan.	El resultado útil es un artefacto: reporte, scorecard y decisión.
La plataforma importa.	Flags, exposición, métricas, warehouse y análisis forman parte del experimento.
El MDE evita autoengaños.	Una muestra pequeña puede aprender mucho y decidir poco.
El rollout también es causalidad aplicada.	Publicar progresivamente protege calidad, coste y operación.
El plan evita cambiar la pregunta.	Hipótesis, métrica, ventanas y reglas deben existir antes del resultado.
Los clústeres cambian el diseño.	Si hay aprendizaje compartido, la unidad individual puede engañar.
Las métricas necesitan contrato.	Nombre, unidad, ventana y dirección deben ser inequívocos.
El tamaño efectivo no siempre coincide con el número de filas.	Si hay dependencia dentro de equipos, colas o aulas, el efecto de diseño reduce la confianza real.
Los DAGs evitan ajustes peligrosos.	Confounder, mediador y collider piden decisiones distintas sobre columnas y filtros.
Los datos observacionales necesitan refutadores.	Placebo, control negativo, subsets y solapamiento ayudan a no creer demasiado pronto.
Los métodos avanzados no sustituyen supuestos.	IPW, AIPW, DID, regression discontinuity e instrumentos solo valen bajo condiciones defendibles.

Para saber más

Angrist, J. D., Imbens, G. W. y Rubin, D. B. (1996). Identification of Causal Effects Using Instrumental Variables. *Journal of the American Statistical Association*, 91(434), 444-455. DOI

Bang, H. y Robins, J. M. (2005). Doubly Robust Estimation in Missing Data and Causal Inference Models. *Biometrics*, 61(4), 962-973. DOI

Benjamini, Y. y Hochberg, Y. (1995). Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B*, 57(1), 289-300. DOI

Austin, P. C. (2009). Balance Diagnostics for Comparing the Distribution of Baseline Covariates Between Treatment Groups in Propensity-Score Matched Samples. *Statistics in Medicine*, 28(25), 3083-3107. DOI

Casella, G. y Berger, R. L. (2002). *Statistical Inference* (2.ª ed.). Duxbury.

CausalML. (2026). CausalML Documentation. Documentación

Card, D. y Krueger, A. B. (1994). Minimum Wages and Employment: A Case Study of the Fast-Food Industry in New Jersey and Pennsylvania. *The American Economic Review*, 84(4), 772-793. JSTOR

Chernozhukov, V. y otros (2018). Double/Debiased Machine Learning for Treatment and Structural Parameters. *The Econometrics Journal*, 21(1), C1-C68. DOI

Chow, S.-C., Shao, J. y Wang, H. (2008). *Sample Size Calculations in Clinical Research* (2.ª ed.). Chapman and Hall/CRC.

Deng, A., Xu, Y., Kohavi, R. y Walker, T. (2013). Improving the Sensitivity of Online Controlled Experiments by Utilizing Pre-Experiment Data. *WSDM*, 123-132. PDF

Donner, A. y Klar, N. (2000). *Design and Analysis of Cluster Randomization Trials in Health Research*. Arnold.

Eppo. (2026). The Eppo Docs. Documentación

Evidently AI. (2026). Evidently Documentation. Documentación

GrowthBook. (2026). GrowthBook Documentation. Documentación

Gupta, S., Ulanova, L., Bhardwaj, S., Dmitriev, P., Raff, P. y Fabijan, A. (2018). The Anatomy of a Large-Scale Experimentation Platform. *IEEE International Conference on Software Architecture*. Microsoft Research

Holland, P. W. (1986). Statistics and Causal Inference. *Journal of the American Statistical Association*, 81(396), 945-960. DOI

Horvitz, D. G. y Thompson, D. J. (1952). A Generalization of Sampling Without Replacement From a Finite Universe. *Journal of the American Statistical Association*, 47(260), 663-685. DOI

Imbens, G. W. y Rubin, D. B. (2015). *Causal Inference for Statistics, Social, and Biomedical Sciences: An Introduction*. Cambridge University Press. DOI

Johari, R., Pekelis, L. y Walsh, D. J. (2017). Peeking at A/B Tests: Why It Matters, and What to Do About It. *KDD*, 1517-1525. DOI

Kish, L. (1965). *Survey Sampling*. Wiley.

Kohavi, R., Longbotham, R., Sommerfield, D. y Henne, R. M. (2009). Controlled Experiments on the Web: Survey and Practical Guide. *Data Mining and Knowledge Discovery*, 18(1), 140-181. Microsoft Research

LaunchDarkly. (2026). Experimentation. Documentación

Lee, D. S. y Lemieux, T. (2010). Regression Discontinuity Designs in Economics. *Journal of Economic Literature*, 48(2), 281-355. DOI

Nie, K., Zhang, Z., Xu, B. y Yuan, T. (2022). Ensure A/B Test Quality at Scale with Automated Randomization Validation and Sample Ratio Mismatch Detection. *CIKM*. arXiv

O'Brien, P. C. y Fleming, T. R. (1979). A Multiple Testing Procedure for Clinical Trials. *Biometrics*, 35(3), 549-556. DOI

OpenFeature. (2026). Evaluation Context. Especificación

OpenFeature. (2026). Introduction. Documentación

Optimizely. (2026). Introduction to Optimizely Feature Experimentation. Documentación

Pearl, J. (2009). *Causality: Models, Reasoning, and Inference* (2.ª ed.). Cambridge University Press.

PostHog. (2026). Creating an Experiment. Documentación

Pocock, S. J. (1977). Group Sequential Methods in the Design and Analysis of Clinical Trials. *Biometrika*, 64(2), 191-199. DOI

PyMC. (2026). Bayesian Non-parametric Causal Inference. Documentación

PyWhy. (2026). DoWhy documentation. Documentación

PyWhy. (2026). EconML documentation. Documentación

Rosenbaum, P. R. y Rubin, D. B. (1983). The Central Role of the Propensity Score in Observational Studies for Causal Effects. *Biometrika*, 70(1), 41-55. DOI

Rubin, D. B. (1974). Estimating Causal Effects of Treatments in Randomized and Nonrandomized Studies. *Journal of Educational Psychology*, 66(5), 688-701. DOI

Sharma, A. y Kiciman, E. (2020). DoWhy: An End-to-End Library for Causal Inference. arXiv

Statsig. (2026). Experiment Options. Documentación

Notas

1. Rubin, D. B. (1974). Estimating Causal Effects of Treatments in Randomized and Nonrandomized Studies. *Journal of Educational Psychology*, 66(5), 688-701. https://doi.org/10.1037/h0037350 ↩

2. Holland, P. W. (1986). Statistics and Causal Inference. *Journal of the American Statistical Association*, 81(396), 945-960. https://doi.org/10.1080/01621459.1986.10478354 ↩

3. Pearl, J. (2009). *Causality: Models, Reasoning, and Inference* (2.ª ed.). Cambridge University Press. ↩

4. Kohavi, R., Longbotham, R., Sommerfield, D. y Henne, R. M. (2009). Controlled Experiments on the Web: Survey and Practical Guide. *Data Mining and Knowledge Discovery*, 18(1), 140-181. https://doi.org/10.1007/s10618-008-0114-1 ↩

5. Gupta, S., Ulanova, L., Bhardwaj, S., Dmitriev, P., Raff, P. y Fabijan, A. (2018). The Anatomy of a Large-Scale Experimentation Platform. *IEEE International Conference on Software Architecture*. https://www.microsoft.com/en-us/research/publication/the-anatomy-of-a-large-scale-experimentation-platform/ ↩

6. OpenFeature. (2026). *Introduction*. https://openfeature.dev/docs/reference/intro/. Consultado el 7 de junio de 2026. ↩

7. OpenFeature. (2026). *Evaluation Context*. https://openfeature.dev/specification/sections/evaluation-context/. Consultado el 7 de junio de 2026. ↩

8. LaunchDarkly. (2026). *Experimentation*. https://launchdarkly.com/docs/home/experimentation. Consultado el 7 de junio de 2026. ↩

9. GrowthBook. (2026). *GrowthBook Documentation*. https://docs.growthbook.io/. Consultado el 7 de junio de 2026. ↩

10. Statsig. (2026). *Experiment Options*. https://docs.statsig.com/statsig-warehouse-native/features/experiment-options. Consultado el 7 de junio de 2026. ↩

11. Eppo. (2026). *The Eppo Docs*. https://docs.geteppo.com/. Consultado el 8 de junio de 2026. ↩

.2. PostHog. (2026). *Creating an experiment*. https://posthog.com/docs/experiments/creating-an-experiment. Consultado el 22 de junio de 2026. ↩

.3. Optimizely. (2026). *Introduction to Optimizely Feature Experimentation*. https://docs.developers.optimizely.com/feature-experimentation/docs/introduction. Consultado el 8 de junio de 2026. ↩

.4. Evidently AI. (2026). *Evidently Documentation*. https://docs.evidentlyai.com/introduction. Consultado el 22 de junio de 2026. ↩

.5. Donner, A. y Klar, N. (2000). *Design and Analysis of Cluster Randomization Trials in Health Research*. Arnold. ↩

.6. Kish, L. (1965). *Survey Sampling*. Wiley. ↩

.7. Casella, G. y Berger, R. L. (2002). *Statistical Inference* (2.ª ed.). Duxbury. ↩

.8. Nie, K., Zhang, Z., Xu, B. y Yuan, T. (2022). Ensure A/B Test Quality at Scale with Automated Randomization Validation and Sample Ratio Mismatch Detection. *CIKM*. https://doi.org/10.1145/3511808.3557087 ↩

.9. Austin, P. C. (2009). Balance diagnostics for comparing the distribution of baseline covariates between treatment groups in propensity-score matched samples. *Statistics in Medicine*, 28(25), 3083-3107. https://doi.org/10.1002/sim.3697 ↩

0. Chow, S.-C., Shao, J. y Wang, H. (2008). *Sample Size Calculations in Clinical Research* (2.ª ed.). Chapman and Hall/CRC. ↩

1. Statsig. (2026). *Experiment Options*. https://docs.statsig.com/statsig-warehouse-native/features/experiment-options. Consultado el 7 de junio de 2026. ↩

2. Johari, R., Pekelis, L. y Walsh, D. J. (2017). Peeking at A/B Tests: Why It Matters, and What to Do About It. *KDD*, 1517-1525. https://doi.org/10.1145/3097983.3097992 ↩

3. Pocock, S. J. (1977). Group Sequential Methods in the Design and Analysis of Clinical Trials. *Biometrika*, 64(2), 191-199. https://doi.org/10.1093/biomet/64.2.191 ↩

4. O'Brien, P. C. y Fleming, T. R. (1979). A Multiple Testing Procedure for Clinical Trials. *Biometrics*, 35(3), 549-556. https://doi.org/10.2307/2530245 ↩

5. Benjamini, Y. y Hochberg, Y. (1995). Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B*, 57(1), 289-300. https://doi.org/10.1111/j.2517-6161.1995.tb02031.x ↩

6. Deng, A., Xu, Y., Kohavi, R. y Walker, T. (2013). Improving the Sensitivity of Online Controlled Experiments by Utilizing Pre-Experiment Data. *WSDM*, 123-132. https://doi.org/10.1145/2433396.2433413 ↩

27. Microsoft Research. (2022). Deep Dive Into Variance Reduction. https://www.microsoft.com/en-us/research/articles/deep-dive-into-variance-reduction/ ↩

28. Sharma, A. y Kiciman, E. (2020). DoWhy: An End-to-End Library for Causal Inference. *arXiv:2011.04216*. https://arxiv.org/abs/2011.04216 ↩

29. PyWhy. (2026). *DoWhy documentation*. https://www.pywhy.org/dowhy/main/index.html. Consultado el 7 de junio de 2026. ↩

30. PyWhy. (2026). *EconML documentation*. https://www.pywhy.org/EconML/spec/overview.html. Consultado el 7 de junio de 2026. ↩

31. Rosenbaum, P. R. y Rubin, D. B. (1983). The Central Role of the Propensity Score in Observational Studies for Causal Effects. *Biometrika*, 70(1), 41-55. https://doi.org/10.1093/biomet/70.1.41 ↩

32. Imbens, G. W. y Rubin, D. B. (2015). *Causal Inference for Statistics, Social, and Biomedical Sciences: An Introduction*. Cambridge University Press. https://doi.org/10.1017/CBO9781139025751 ↩

33. Horvitz, D. G. y Thompson, D. J. (1952). A Generalization of Sampling Without Replacement From a Finite Universe. *Journal of the American Statistical Association*, 47(260), 663-685. https://doi.org/10.1080/01621459.1952.10483446 ↩

34. Bang, H. y Robins, J. M. (2005). Doubly Robust Estimation in Missing Data and Causal Inference Models. *Biometrics*, 61(4), 962-973. https://doi.org/10.1111/j.1541-0420.2005.00377.x ↩

35. Card, D. y Krueger, A. B. (1994). Minimum Wages and Employment: A Case Study of the Fast-Food Industry in New Jersey and Pennsylvania. *The American Economic Review*, 84(4), 772-793. https://www.jstor.org/stable/2118030 ↩

36. Lee, D. S. y Lemieux, T. (2010). Regression Discontinuity Designs in Economics. *Journal of Economic Literature*, 48(2), 281-355. https://doi.org/10.1257/jel.48.2.281 ↩

37. Angrist, J. D., Imbens, G. W. y Rubin, D. B. (1996). Identification of Causal Effects Using Instrumental Variables. *Journal of the American Statistical Association*, 91(434), 444-455. https://doi.org/10.1080/01621459.1996.10476902 ↩

38. PyWhy. (2026). *DoWhy documentation*. https://www.pywhy.org/dowhy/main/index.html. Consultado el 7 de junio de 2026. ↩

39. Sutton, R. S. y Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2.ª ed.). MIT Press. https://incompleteideas.net/book/the-book-2nd.html ↩

40. CausalML. (2026). *CausalML Documentation*. https://causalml.readthedocs.io/. Consultado el 22 de junio de 2026. ↩

1. PyMC. (2026). *Bayesian Non-parametric Causal Inference*. https://www.pymc.io/projects/examples/en/latest/causal_inference/bayesian_nonparametric_causal.html. Consultado el 22 de junio de 2026. ↩

Capítulo 08: Recapitulación de ciencia de datos

Entrando en el cierre

Este capítulo no introduce una técnica aislada. Sirve para comprobar si el facsímil entero se ha convertido en criterio de ingeniería. Después de hablar de contratos, calidad, splits, embeddings, slices, operación y causalidad, la pregunta ya no es “qué métrica sale”, sino “qué decisión puedo defender con lo que sé”.

Ese matiz cambia la forma de trabajar. Un dataset puede tener columnas correctas y aun así no estar listo. Un experimento puede estar bien planteado y aun así no tener muestra suficiente. Un sistema puede mejorar la media global y empeorar justo el segmento que más nos importa. Por eso el cierre del facsímil no pide una respuesta decorativa: pide reunir evidencias, ejecutar un gate, escribir una decisión y dejar claro qué se corrige antes de publicar.

Qué deberías poder hacer al terminar

Este facsímil empezó con una idea sencilla: en IA, los datos no son un trámite. Son parte del sistema. Si el dataset está mal definido, si el contrato no existe, si el split engaña, si los slices fallan, si falta trazabilidad o si confundimos predicción con intervención, el modelo puede parecer sofisticado y aun así tomar malas decisiones.

Al cerrar el facsímil deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Auditar un dataset antes de usarlo.	Revisas contrato, columnas, valores, licencias, linaje y trazabilidad.
Diseñar una evaluación honesta.	Separas train, validation y test sin leakage.
Leer features y embeddings como decisiones.	No conviertes columnas en vectores sin contrato.
Medir slices críticos.	No publicas una media global si falla un segmento importante.
Operar datos en producción.	Escribes SLIs, SLOs, runbooks, trazas y gates.
Diseñar un experimento aplicable.	Separas predicción de intervención, ATE de CATE y A/B de observacional.
Entregar una decisión defendible.	Generas scorecard, reporte y documento de salida.

Esta tabla no está pensada como una lista de objetivos de un temario. Es más útil leerla como una pequeña prueba de realidad. Si una persona te da un CSV, un modelo, una métrica y una fecha de publicación, ¿qué preguntas harías antes de aceptar la release? ¿Qué pedirías ver? ¿Qué parte podrías automatizar y qué parte exigiría revisión humana? En ciencia de datos aplicada a IA, saber programar una comprobación es solo la mitad; la otra mitad es saber cuándo esa comprobación cambia una decisión.

La diferencia con una práctica puramente académica está ahí. En una práctica puedes aprobar con una respuesta correcta. En un sistema real tienes que dejar rastro de por qué esa respuesta era defendible en ese momento, con esos datos, esos límites y esa evidencia. Por eso este cierre insiste tanto en contratos, trazas, slices, gates y experimentos. Son palabras poco vistosas, pero sostienen el trabajo cuando el proyecto deja de ser una demo.

La frase de cierre:

> La ciencia de datos en IA no consiste en encontrar una métrica bonita. Consiste en construir una decisión que aguante preguntas.

Lo que hemos construido

El recorrido del facsímil puede leerse como una cadena:

La cadena no es una ocurrencia editorial. Datasheets for Datasets consolidó la idea de documentar motivación, composición, recogida, usos y límites de un dataset antes de ponerlo a circular.[1] Model Cards hizo algo parecido para modelos, obligándonos a declarar uso previsto, métricas, grupos evaluados y limitaciones.[2] Y el ML Test Score recordaba una cosa muy de ingeniería: un sistema de ML no está maduro solo porque tenga una métrica alta; necesita tests de datos, monitorización, reproducibilidad y gestión de cambios.[3]

CAPÍTULO	PREGUNTA	ARTEFACTO MENTAL
01	¿Qué dato tenemos y de dónde viene?	Contrato, linaje y dataset card.
02	¿El dato cumple lo mínimo?	Gate de calidad.
03	¿Medimos sin engañarnos?	Split y manifiesto de evaluación.
04	¿Cómo representamos la información?	Features, embeddings y búsqueda.
05	¿A quién falla la decisión?	Slices, políticas y mitigación.
06	¿Qué pasa cuando llega producción?	DataOps, drift, trazas y postmortem.
07	¿Una acción cambia algo?	Experimento, causalidad y rollout.

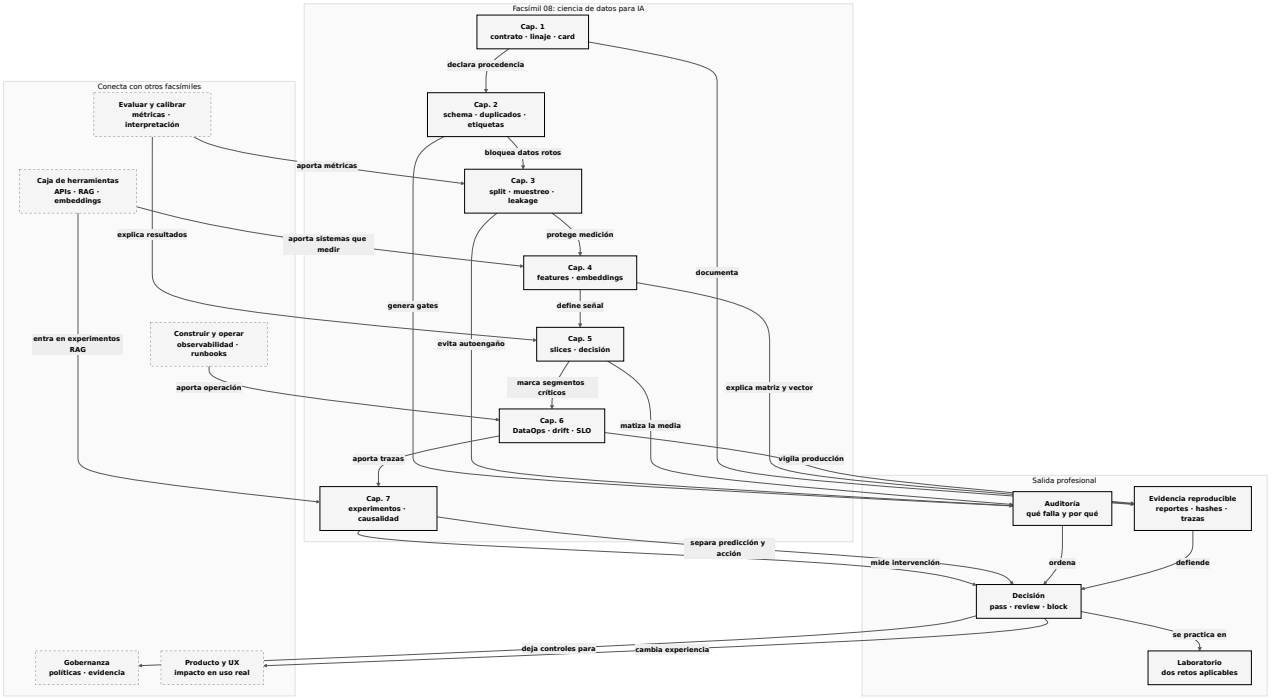
Leída así, la ciencia de datos para IA deja de ser una colección de técnicas y se convierte en una disciplina de evidencia. Cada capítulo añade una defensa contra una forma distinta de autoengaño: usar datos sin permiso, limpiar sin contrato, medir con leakage, vectorizar sin trazabilidad, publicar una media global, operar sin runbook o llamar causal a una correlación. La cadena completa importa porque el sistema falla por el eslabón más débil, no por el capítulo que mejor entendimos.

También cambia la manera de estudiar el facsímil. No hace falta memorizar cada archivo ni cada paso. Lo importante es reconocer el patrón: antes de usar datos, declaro qué son; antes de evaluar, protejo la separación; antes de vectorizar, entiendo qué señal estoy fabricando; antes de publicar, miro a quién falla; antes de escalar, observo producción; y antes de afirmar impacto, diseño una intervención medible. Si ese patrón queda dentro, el lector puede llevarlo a otro dominio aunque cambien los nombres de columnas, proveedores o herramientas.

Si una persona termina este facsímil y solo recuerda una cosa, que sea esta: **cada métrica necesita una historia de procedencia, una unidad de análisis, una ventana, un contrato y una decisión permitida**.

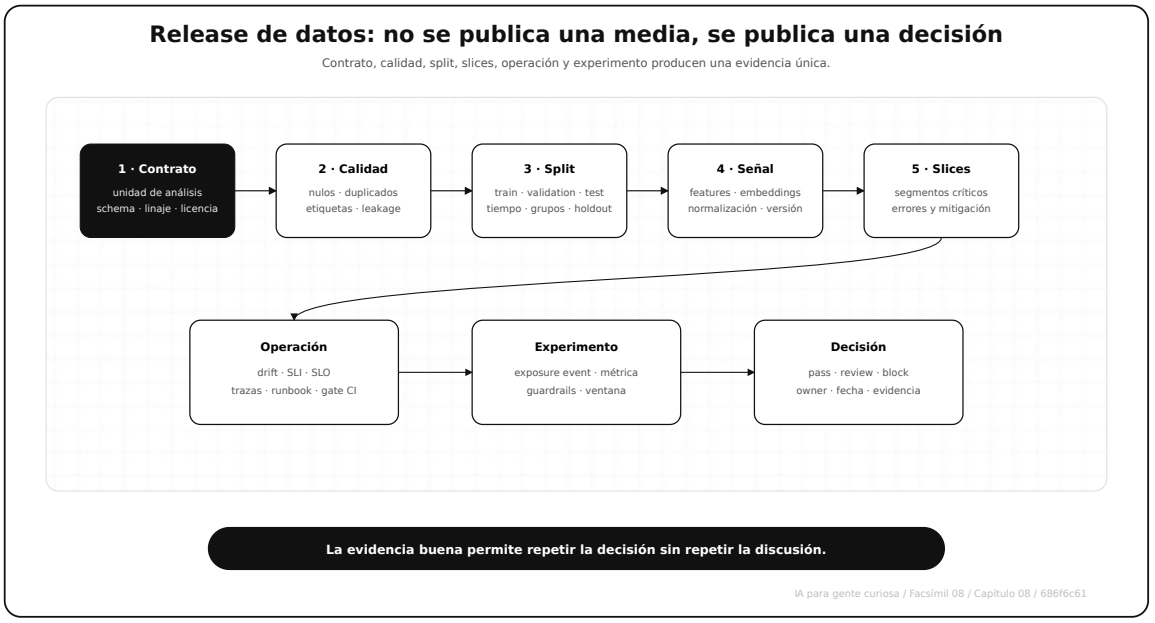
Cómo encaja todo

Este mapa une el facsímil completo. No intenta repetir cada tabla, sino mostrar el flujo profesional: del dato inicial a la decisión de publicación. La lectura buena no es lineal del todo: contrato, calidad, split, representación, slices, operación y causalidad se revisan entre sí.



Anatomía visual: de dataset a decisión

El cierre del facsímil necesitaba una figura propia porque la ciencia de datos no termina en el dataframe. Termina cuando alguien puede explicar qué dato entró, qué se midió, qué se corrigió, qué quedó fuera y qué decisión se tomó.



La revisión de datos une contrato, calidad, medición, segmentos, operación y experimento. El resultado útil no es “la métrica sube”, sino una decisión trazable.

En el día a día: quién mira qué

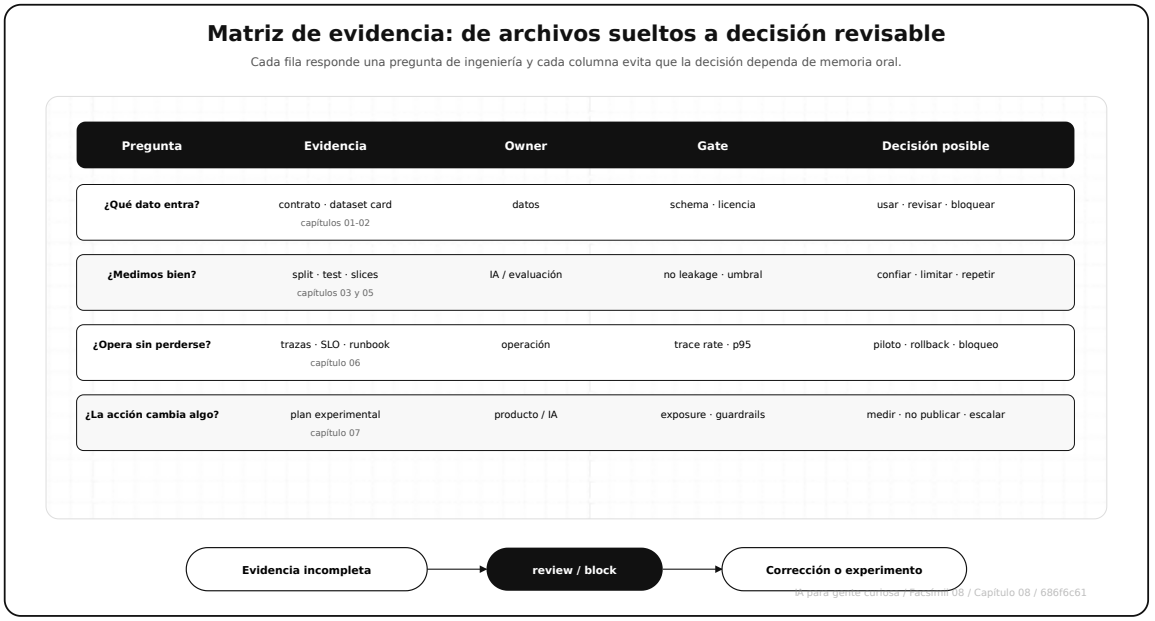
En una práctica universitaria puede parecer que todo termina cuando el script imprime `block` o `review`. En un equipo real, ese es solo el principio de la conversación. Una persona de datos mirará si el contrato y el linaje son defendibles. Una persona de IA mirará si el split, la métrica y los slices permiten confiar en la evaluación. Una persona de producto preguntará si el fallo afecta a usuarios reales y qué alcance se puede permitir. Operación preguntará si hay trazas, SLOs y rollback. Y una persona con responsabilidad de cumplimiento o gobernanza preguntará si la evidencia se conserva, si hay owner y si la decisión se puede reconstruir dentro de tres meses.

Esta separación de responsabilidades no es burocracia. Amershi y otros documentaron que los sistemas de aprendizaje automático introducen fricciones específicas en requisitos, datos, evaluación, monitorización y coordinación entre perfiles.[4] El NIST AI RMF insiste en una idea compatible: gestionar riesgo de IA exige mapear, medir, gestionar y gobernar, no solo optimizar una métrica.[5] Traducido a este facsímil: cada evidencia debe tener una pregunta, un archivo, un responsable y una decisión posible.

Por eso no basta con entregar outputs. El alumno debería poder decir: “esta evidencia sostiene este check; este check permite esta decisión; esta decisión deja este riesgo residual; y este riesgo define el siguiente experimento”. Cuando esa cadena existe, el trabajo se puede revisar. Cuando no existe, solo tenemos una carpeta con archivos.

Anatomía visual: matriz de evidencia de release

Esta figura añade la parte que suele faltar cuando se habla de ciencia de datos de forma demasiado abstracta: el expediente. Un expediente de release no es una carpeta enorme; es una matriz donde cada fila responde a una pregunta técnica y cada columna obliga a conectar evidencia, owner y decisión.



> La matriz obliga a que cada pregunta tenga evidencia, owner, gate y decisión. Así el trabajo deja de ser una lista de archivos y se convierte en una revisión profesional.

Cuadernos para practicar

Has cuidado los datos a lo largo del facsímil; estos cuadernos te dejan ver, en directo, las dos formas más comunes de engañarte con ellos. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar. Cada uno, explicado paso a paso, con salidas reales, y anclado al capítulo del que sale.

Leakage: cómo te engañas sin querer

Qué practicas: provocar y detectar la fuga de datos, el error que infla resultados sin dar error. **Dónde encaja:** capítulos 2 y 3 (calidad de datos y *leakage*; *splits* y medir sin engañarse). **Qué necesitas:** un navegador. Corre en CPU; sin claves.

Montas el escenario más honesto posible: 120 muestras, 1000 columnas de ruido y una etiqueta totalmente aleatoria. No hay nada que aprender. Y aun así consigues que un modelo presuma de un **77,5%** de acierto... haciendo trampa sin querer: seleccionas las «mejores» columnas mirando el dataset entero *antes* de partir en train y test. Cuando lo haces bien —el test escondido hasta el final, la selección dentro de la validación— el número cae al **45%**, el del puro azar. La fuga inflaba **32 puntos** sobre datos sin señal alguna.

Casi todos los «modelos increíbles» que fracasan al desplegarse tenían fuga. Es silenciosa: no da error, da buenas noticias falsas.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Slices y sesgo: el promedio miente

Qué practicas: descubrir que un buen acierto global puede esconder un fallo grave en un subgrupo. **Dónde encaja:** capítulo 5 (slices, sesgos y decisión algorítmica). **Qué necesitas:** un navegador. Corre en CPU; sin claves.

Entrenas un modelo sobre dos grupos: uno mayoritario (85%) y otro minoritario (15%) con un patrón distinto. El acierto global sale en un tranquilizador **83%**. Pero cuando partes por grupos, el promedio confiesa: el mayoritario va al **92%** y el minoritario se desploma al **36%**, peor que lanzar una moneda. El global se parecía al del grupo grande porque son la mayoría de los casos y mandan en la media; el pequeño casi no la mueve... pero recibe un servicio mucho peor.

Un modelo que va «bien de media» puede ser inútil o injusto para un colectivo concreto. Si decides sobre personas, el promedio no te exime.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Vocabulario aprendido

Estas palabras no son adorno. Son el vocabulario mínimo para discutir el trabajo con alguien de datos, IA, producto u operación sin que todo se convierta en “me parece que”. Si puedes usar estos términos para explicar tu entrega, probablemente has entendido el cierre del facsímil.

TÉRMINO	QUÉ SIGNIFICA AQUÍ	CÓMO LO USARÍAS EN UNA ENTREGA
Auditoría de datos	Revisión sistemática de contrato, calidad, trazabilidad, splits, slices y decisión.	La usas para ordenar qué se comprueba antes de publicar o automatizar.
Decisión de publicación	Salida que indica si un sistema puede pasar a uso real, revisión o bloqueo.	La escribes como una decisión defendible, no como una opinión rápida.
Decisión de release de datos	Dictamen técnico sobre si los datos y sus derivados permiten automatizar, revisar o bloquear.	Lo escribes como `pass`, `review` o `block`, con checks y evidencia.
Evidencia reproducible	Reportes, hashes, manifests, trazas, contratos y salidas que otra persona puede regenerar.	La adjuntas al memo para que la revisión no dependa de confianza verbal.
Gate de datos	Regla ejecutable que impide publicar si fallan calidad, split, trazabilidad, slices o SLOs.	Lo conectas a CI o a la revisión de release.
Expediente de release	Conjunto de contrato, datos, evidencias, outputs, memo y decisión que permite revisar una publicación de datos o IA.	Lo preparas para que otra persona pueda reconstruir el porqué de la decisión.
Alcance limitado	Publicación o piloto restringido a un segmento, uso o ventana porque la evidencia todavía no permite un despliegue general.	Lo propones cuando no hay bloqueo, pero la señal aún no permite rollout general.
Owner de evidencia	Persona o equipo responsable de mantener, explicar y actualizar una evidencia concreta del expediente.	Lo asignas para que un check no quede huérfano cuando cambie el dato o el sistema.
Corrección sin relajar umbrales	Arreglar datos, trazas o contratos sin bajar el listón después de ver el fallo.	Mantienes el contrato y corriges la causa, no la métrica incómoda.
Siguiente experimento	Intervención medible para aprender si una acción mejora el sistema en un slice concreto.	Diseñas unidad, exposición, métrica primaria, guardrails y criterio de parada.
Laboratorio	Espacio guiado donde conviertes el facsímil en trabajo práctico.	Lo usas para producir artefactos que puedas enseñar, ejecutar y defender.

Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Querer publicar por una métrica global	La media resume demasiado.	Mirar contrato, test, slices y operación.
Arreglar el modelo antes de arreglar datos	Parece la parte más interesante.	Revisar trazabilidad y calidad primero.
Diseñar experimentos sin exposure event	La asignación parece suficiente.	Registrar exposición real y versión exacta.
Ignorar ventanas de maduración	Queremos decidir rápido.	Declarar `day_1`, `day_7` o la ventana que toque.

Antes de pasar página

Antes de cerrar el facsímil, deberías poder responder:

1. ¿Qué diferencia hay entre contrato de datos y contrato de experimento?
2. ¿Por qué un dataset con schema correcto puede quedar bloqueado?
3. ¿Qué hace que un split sea honesto?
4. ¿Por qué los slices críticos pueden cambiar una decisión?
5. ¿Qué SLI de DataOps bloquearía una publicación?
6. ¿Qué diferencia hay entre asignación y exposición?
7. ¿Por qué un experimento en `review` no es necesariamente un fracaso?
8. ¿Qué entregarías como evidencia reproducible de una decisión?
9. ¿Por qué no deberías cambiar umbrales después de mirar el resultado?
10. ¿Qué diferencia hay entre corregir trazabilidad y mejorar el modelo?
11. ¿Qué comprobaría un gate de CI de datos?

En resumen

Este cierre no pretende que salgas pensando que la ciencia de datos es una lista infinita de controles. Pretende justo lo contrario: que tengas una forma ordenada de no perderte. Cuando un proyecto crece, siempre aparecen urgencias, atajos y métricas seductoras. El

contrato, la evidencia, el gate y el experimento son la manera de seguir pensando con calma cuando el sistema ya tiene presión real.

IDEA	QUÉ TE LLEVAS
Los datos son sistema.	No son entrada pasiva del modelo.
La evaluación es una promesa.	Si el split o los slices fallan, la métrica no basta.
La operación decide.	Sin trazas, SLOs y runbooks, no hay publicación responsable.
La causalidad cambia la pregunta.	Predecir no prueba que una acción funcione.
El cierre junta todo.	La salida profesional es una decisión reproducible, no una impresión.

Recursos para seguir: leer, construir y experimentar

La ciencia de datos es la base callada de todo lo demás: sin datos con linaje, sin splits honestos y sin mirar por slices, ningún modelo es de fiar. Aquí tienes por dónde seguir.

Para experimentar sin código. En las cajas «Pruébalo en 5 minutos» lo viste: Teachable Machine (`teachablemachine.withgoogle.com`) para provocar una fuga de datos con tus propias fotos; el Embedding Projector (`projector.tensorflow.org`) para ver cómo las features se vuelven una estructura visible con PCA y t-SNE; y el What-If Tool de Google (`pair-code.github.io/what-if-tool`) para descubrir que un buen promedio puede esconder un grupo al que el modelo trata peor.

Para construir. Las herramientas de trabajo diario: pandas para manipular datos, scikit-learn para modelos y validación, Fairlearn para medir y mitigar sesgos por subgrupo, y herramientas de calidad de datos como Great Expectations para validar esquemas y detectar problemas antes de entrenar.

Para leer. Cada capítulo enlaza sus fuentes en «Para saber más»; las ideas que más te ahorrarán disgustos son el leakage (el error que infla métricas sin que te enteres) y la evaluación por slices. Recorrer el ciclo entero, de los datos crudos a una decisión, es lo que separa «el número salió bonito» de «mi evaluación no me está engañando».

Para saber más

Amershi, S. y otros (2019). Software engineering for machine learning: A case study. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

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. *2017 IEEE International Conference on Big Data*, 1123-1132. https://doi.org/10.1109/BigData.2017.8258038

Gebru, T. y otros (2021). Datasheets for datasets. *Communications of the ACM*, 64(12), 86-92. https://doi.org/10.1145/3458723

Kohavi, R., Longbotham, R., Sommerfield, D. y Henne, R. M. (2009). Controlled experiments on the web: Survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1), 140-181. https://doi.org/10.1007/s10618-008-0114-1

Mitchell, M. y otros (2019). Model cards for model reporting. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220-229. https://doi.org/10.1145/3287560.3287596

National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. https://doi.org/10.6028/NIST.AI.100-1

Sambasivan, N. y otros (2021). “Everyone wants to do the model work, not the data work”: Data cascades in high-stakes AI. *Proceedings of CHI 2021*, 1-15. https://doi.org/10.1145/3411764.3445518

Sculley, D. y otros (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems 28*. https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems

---

Notas

1. Gebru, T. y otros (2021). Datasheets for datasets. *Communications of the ACM*, 64(12), 86-92. https://doi.org/10.1145/3458723 ↩

2. Mitchell, M. y otros (2019). Model cards for model reporting. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220-229. https://doi.org/10.1145/3287560.3287596 ↩

3. 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. *2017 IEEE International Conference on Big Data*, 1123-1132. https://doi.org/10.1109/BigData.2017.8258038 ↩

4. Amershi, S. y otros (2019). Software engineering for machine learning: A case study. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042 ↩

5. National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. https://doi.org/10.6028/NIST.AI.100-1 ↩