// WHITE PAPER · PUBLISHED 12 SEPTEMBER 2026

Data Pipelines for Machine Learning: The Duplicate That Hid the Heartbeat

ETL, ELT, publish and subscribe, and the cleaning a model inherits before it learns anything

Jason Newell · MAX Research Collective · Report WPR-DAT-007 · Published 12 September 2026
Licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/
Original: https://www.maxresearchcollective.com/articles/data-pipelines-for-machine-learning.html
Cite as: Newell, J. (2026). Data Pipelines for Machine Learning: The Duplicate That Hid the Heartbeat. MAX Research Collective, WPR-DAT-007. https://www.maxresearchcollective.com/articles/data-pipelines-for-machine-learning.html

Two sensors, one heartbeat

REEF's documentation illustrates its beacon finding with a device that calls one outside address every fifty-eight seconds, give or take three. REEF is a local-first network monitor I run on a small network, and that pattern is what it was built to notice. It reads connection records from the gateway and scores every pair of device and destination for rhythm. People are bursty. A machine checking in with its operator keeps time.

The scoring is plain arithmetic. REEF sorts the start times of the connections and takes the gaps between them; then it divides the standard deviation of those gaps by their mean. That ratio is the coefficient of variation, or CV. It measures how steady a rhythm is, whatever the tempo. A beacon is the regular callback that malware makes to its command-and-control server, and REEF counts a group as one when three things hold: at least six connections, a mean gap between ten seconds and an hour, and a CV at or under 0.15.

Then a second sensor arrived. The gateway's deep packet inspection engine had been the only source of connection records. The firewall's own packet-filter log could supply an independent copy of the same traffic, with no new hardware at all. Two witnesses should beat one. The obvious design was to pour both feeds into one table and let the detector read the lot.

Pour them together and a heartbeat like that one disappears. Each connection now appears twice, once per sensor, a fraction of a second apart. Table 1 puts numbers on it with a constructed example: eight connections roughly fifty-eight seconds apart, with sensor A stamping them to the millisecond and sensor B stamping whole seconds on a clock that runs 0.3 seconds fast. Merged and sorted, the gaps stop reading 58, 58, 58. They read 57, then 0.412, then 58.588, then 0.03. The mean gap halves to 29 seconds, which is still inside the band. The CV climbs from 0.010 to 0.992 — more than six times the ceiling — and the detector looks at a device keeping near-perfect time and reports nothing at all.

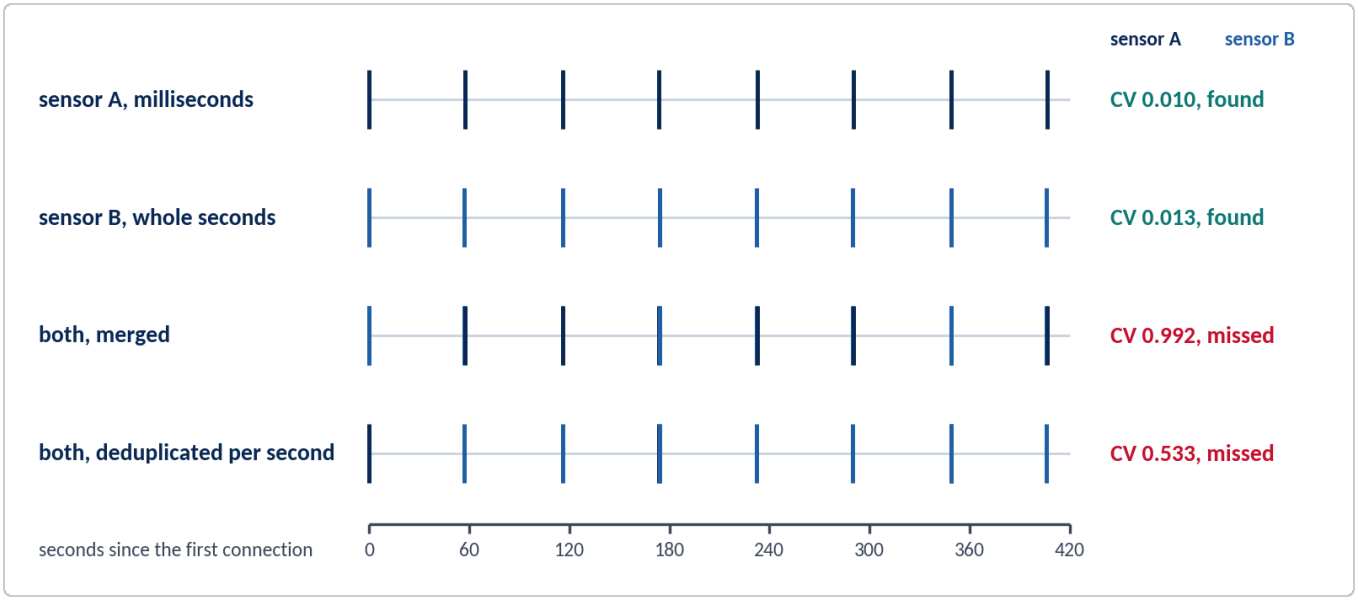


Figure 1. At this scale each pair of marks prints as one mark, so the merged lanes look regular to the eye and not to the arithmetic.

The table reads the two feeds four ways. Each way runs through the same thresholds.

FEED READ BY THE DETECTOR	TIMESTAMPS	GAPS KEPT	MEAN GAP, SECONDS	CV	BEACON FOUND
Sensor A alone	8	7	58.04	0.010	yes
Sensor B alone	8	7	58.00	0.013	yes
Both, merged	16	14	29.02	0.992	no
Both, deduplicated per second	10	9	45.11	0.533	no

Table 1. The same eight connections pass the beacon test from either sensor alone and fail it from both together, deduplicated or not.

The tempting repair is deduplication. It half works. Collapse every record that shares a device, a destination and a whole second, and six of the eight pairs merge. Two survive, because the clocks disagree across a second boundary: 173.905 on sensor A lands as 174 on sensor B. Those survivors put gaps of 0.095 and 0.156 seconds back into the series. The CV falls to 0.533, still more than three times the ceiling.

REEF's pull request that added the second sensor took a different route, and it shipped with a regression test proving the naive merge misses a true beacon. Each feed is scored on its own. A finding that both sensors produce gains a line of evidence saying a second sensor corroborated it. The rows stay apart; only the findings are compared.

Nothing in that story involves a model going wrong. The detector did what arithmetic does with the numbers it was handed. The fault was a decision about the data upstream of it (whether two records were one event, and at which stage to decide), and every stage of a data pipeline makes that decision somewhere, from the tools that move records around to the machine learning that now cleans them and then learns from them. The stages differ mostly in what it costs to take the decision back.

What a pipeline is for

A data pipeline is the set of jobs that move records from where they're produced to where something uses them, changing them on the way. Sources sit at one end: an application database, a vendor's API, a sensor, a log file. Consumers sit at the other: a dashboard, a report, a detector, a model in training. Everything between the two is somebody's decision about shape, timing and trust.

Two timing styles cover most of it. A batch pipeline wakes on a schedule and reads everything that changed since the last run. It processes that as one lump; that suits a nightly revenue report and very little that has to react. A streaming pipeline handles each record as it arrives, or in windows of a few seconds. A beacon detector needs that kind. Most working systems run both: the stream feeds alerts, and a batch job rebuilds history overnight.

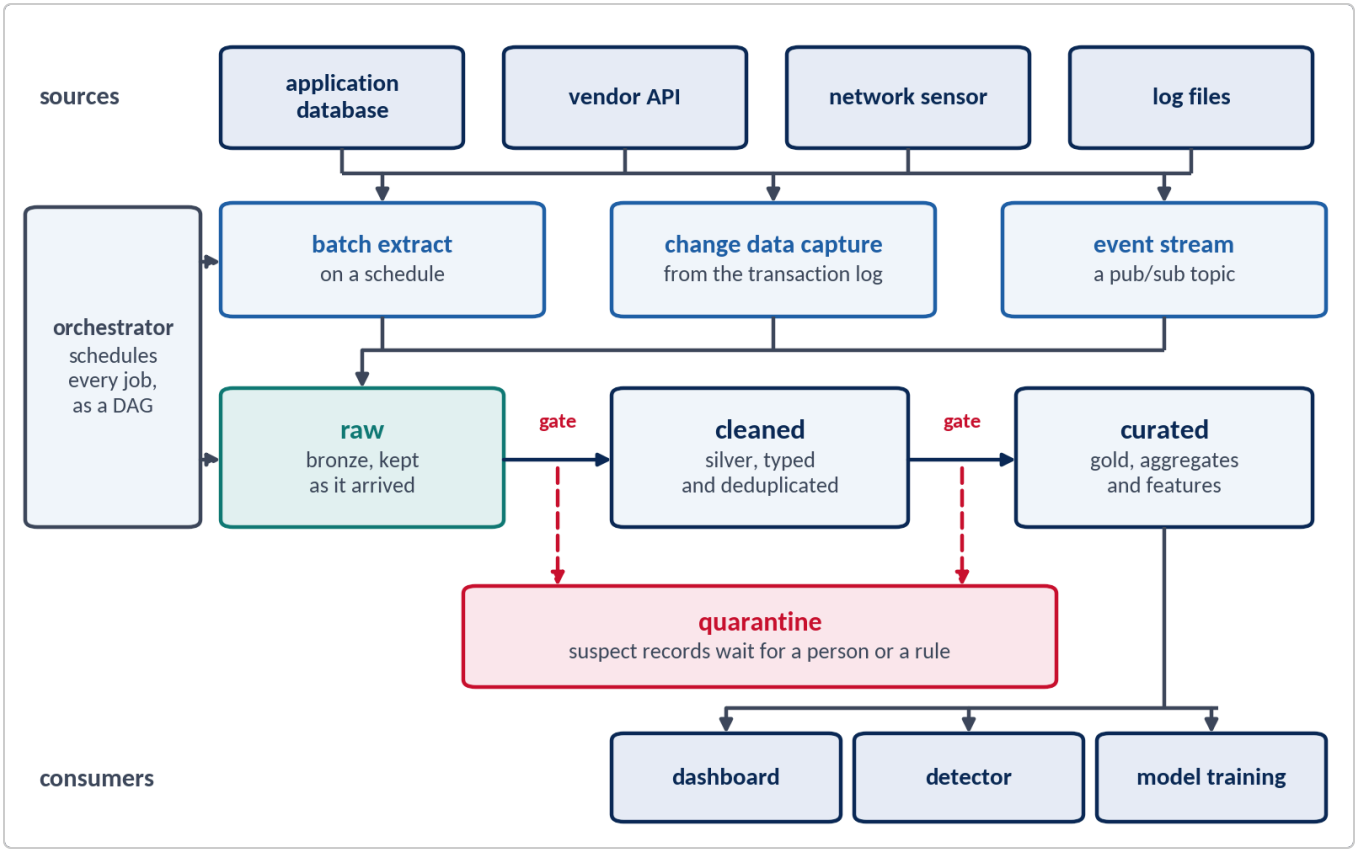


Figure 2. Every layer above raw can be rebuilt from the one beneath it, and the gates decide what moves up and what waits.

Figure 2 draws the stages as layers. Raw data lands first and stays exactly as it arrived. A cleaned layer holds records that have been typed, deduplicated and conformed to shared keys; a curated layer holds the aggregates and features that consumers actually read. Databricks documents the pattern as the medallion architecture, with the layers named bronze, silver and gold. The names matter less than the rule underneath. Any layer above raw can be thrown away and rebuilt by running the jobs again.

Two kinds of machinery sit around the layers. An orchestrator schedules the jobs and knows which one waits for which. It usually holds that as a directed acyclic graph, or DAG, a set of tasks joined by arrows that never loop back on themselves. Quality checks sit on the boundaries between layers. Each one decides whether a batch moves up or goes to quarantine, the holding area where suspect records wait for a person or a rule to release them.

The consumer that matters most here is the model. It's the most demanding consumer a pipeline ever serves. A dashboard shows a wrong number to a person who might notice it; a model folds the wrong number into its parameters and then applies it to every record it sees afterwards. Sculley and colleagues made the point in their 2015 NIPS paper on hidden technical debt in machine learning systems (the conference is now called NeurIPS), whose first figure draws the model code as a small box surrounded by far larger boxes for data collection, data verification, feature extraction and the rest of the machinery a production system needs around it. The picture is an illustration rather than a measurement. Anyone who has run one of these systems recognises it anyway.

ETL and ELT, and where the raw rows live

ETL stands for extract, transform, load. Records are pulled from a source and reshaped on an intermediate server. Only the reshaped result is written to the destination, which is usually a warehouse: a database built for analytical queries over large volumes. The pattern goes back to the 1970s and 1980s, to destinations with little storage and less compute to spare.

ELT swaps the last two letters. Records are extracted and loaded into the destination in their raw form, and the transformation runs afterwards, inside the warehouse, usually as SQL. Google Cloud's documentation describes it in those terms. The pattern spread with cloud warehouses; storage got cheap enough that keeping everything became the default.

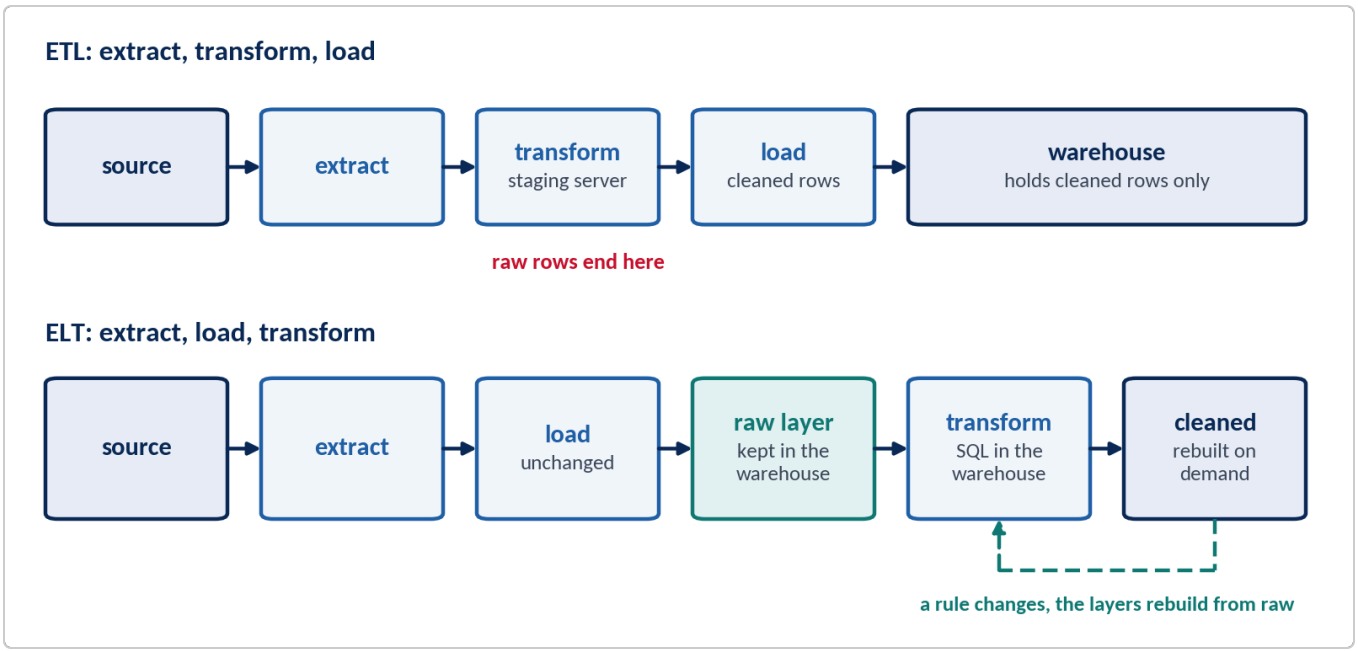


Figure 3. Under ETL the raw rows are gone before anything is stored, and under ELT they stay beneath every cleaning rule.

The textbook comparison is about cost and speed. For machine learning the comparison that counts is reversibility. Under ETL a transformation decision is fixed at the moment of load, so if the transform merged two sensors' rows, rounded timestamps to the second, or dropped every record with a missing field, the warehouse never holds the version from before that happened. The only way back is to extract again — from sources that may have discarded the originals by then. Under ELT the raw rows are still there. A wrong cleaning rule becomes a bug with a fix: change the transform, rebuild the layers above raw, and retrain.

That advantage is largest where cleaning is most likely to be wrong. Early in a model's life nobody knows yet which quirks of the data are noise and which are signal. REEF's merge was recoverable because it was caught before anything learned

from the merged rows. A model trained for six months on those rows would be a harder problem. Only a copy of the rows from before the merge could rescue it.

ETL still earns its place. Some data should never land raw at all, such as a card number, a patient identifier, or a secret somebody pasted into a log line; stripping those before they're stored is often a compliance obligation rather than a preference. A destination that can't afford the compute to transform points the same way, and so does a source so large that only a filtered slice is worth moving. Most working pipelines end up hybrid. A thin transform before load removes what must not persist. The heavy cleaning runs after load, where it can be run again.

QUESTION	ETL	ELT
Where the transform runs	On an intermediate server, before the destination sees the data	Inside the destination, after the raw data has landed
What the destination holds	Cleaned rows only	Raw rows, with cleaned and curated layers built on top
What a wrong cleaning rule costs	A fresh extract from sources that may no longer hold the originals	A changed transform and a rebuild of the layers above raw
Sensitive fields	Removed before anything is stored	Need a thin transform before load, or they land raw
Fits best when	Storage or destination compute is scarce, or data must be filtered before it persists	Storage is cheap and the cleaning rules are still being learned

Table 2. The practical difference between the two patterns is what survives a cleaning rule that turns out to be wrong.

Publish and subscribe, and the log that remembers

Batch pipelines pull their data. Streaming pipelines mostly have it pushed at them, and the pattern underneath is publish and subscribe, or pub/sub. A producer publishes a message to a named channel called a topic without knowing who will read it — or whether anyone will. Subscribers register interest in the topic and receive what gets published. Neither side needs the other's address. That property lets a second detector, an archive job and a model's feature builder all read the same connection records while the sensor stays exactly as it was.

Implementations fall into two families, and the difference decides what a pipeline can do after a mistake. A message queue hands each message to a consumer and forgets it once the consumer acknowledges it. Amazon SQS works this way; Amazon SNS sits in front of queues and fans one published message out to many of them. A log-based system keeps messages for a retention period whether or not anybody has read them. Apache Kafka is the best-known example. It appends records to topics split into partitions, and each consumer tracks its position in each partition as a number called an offset. A consumer can rewind to an old offset and read the history again.

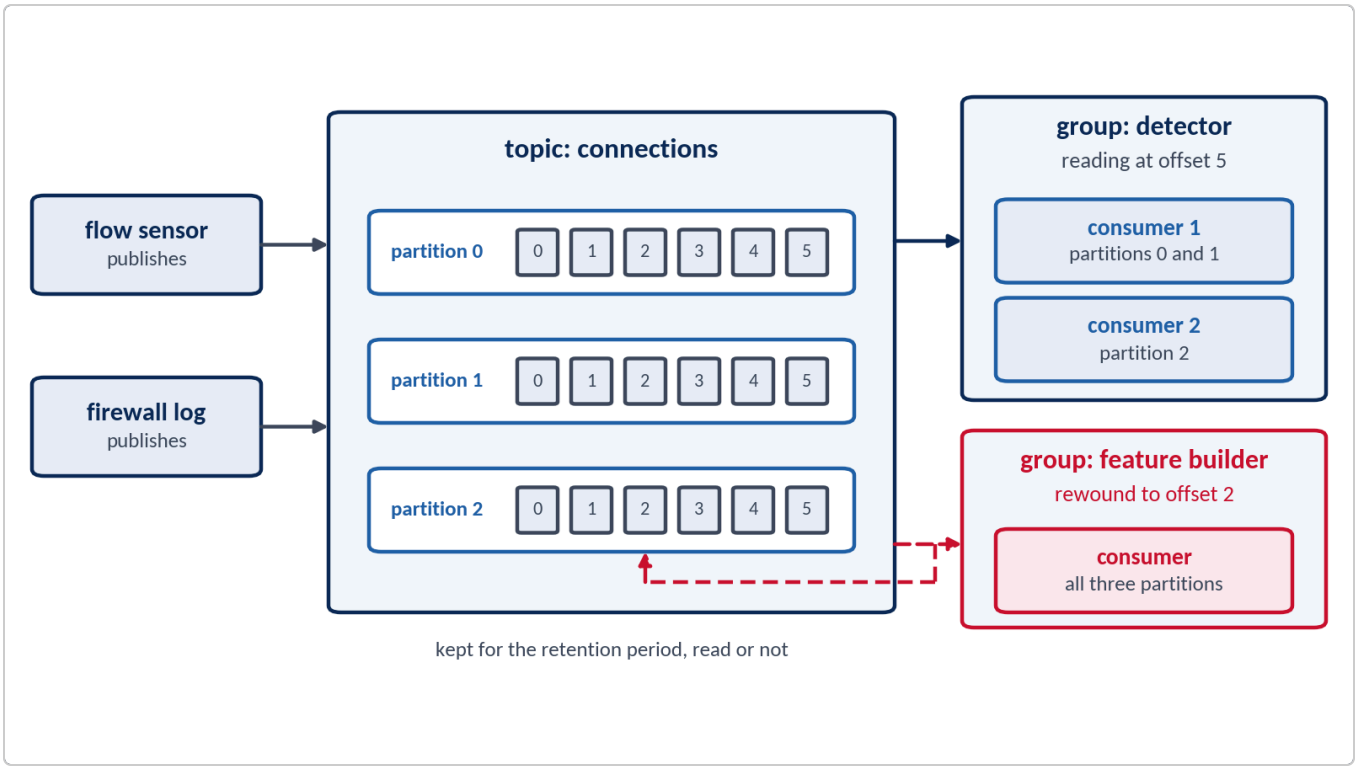


Figure 4. Two consumer groups read one topic at their own offsets, and the group with the wrong rule rewinds without disturbing the other.

Replay is the streaming version of keeping the raw rows. Google Cloud Pub/Sub offers it too: a subscription can seek back to a timestamp or a saved snapshot, and its messages are kept for seven days by default. Figure 4 shows the Kafka shape. Consumers that share a group name split a topic's partitions between them, with each partition read by exactly one member of the group at a time. Order is guaranteed inside a partition. It is guaranteed nowhere else. A detector that needs every record for one device in sequence therefore keys its messages by device, so all of that device's records land in the same partition.

Every broker makes a promise about delivery, and there are three to choose from. At most once means a message can be lost and is never repeated. At least once means it is never lost and can arrive twice; that's the default for most systems, SQS standard queues and Pub/Sub among them. Exactly once is the expensive promise. The scope is always narrow. Kafka has offered it since version 0.11 in 2017 for work that reads from Kafka, processes, and writes back to Kafka, using transactions and idempotent producers (producers whose retries can't create a second copy).

Pub/Sub offers exactly-once delivery on pull subscriptions whose subscribers connect in the same region.

At least once is the honest default. It hands the cleaning problem straight to the consumer. A consumer that might see a message twice has to be idempotent, which means processing the same message a second time changes nothing. REEF's two sensors are that problem in physical form: the network delivered every connection at least once, and the pipeline had to decide what twice meant.

Streams add a clock problem that batch jobs can mostly ignore. Event time is when a thing happened; processing time is when the pipeline got round to it. A sensor that was offline for an hour sends its records an hour late, and a job that files by processing time puts every one of them in the wrong hour. Apache Flink, Apache Beam and Spark Structured Streaming all handle this with watermarks. A watermark is a running declaration that event time has reached some point, so records stamped earlier are no longer expected; late ones can still turn up, and the pipeline's rules decide what happens to them.

REEF's behavioural baseline makes the same choice on a small scale. That's the part that learns what normal looks like for each device. Every flow is bucketed by its own timestamp rather than by the moment REEF happened to poll, so the learned profile doesn't depend on how often anything polls. A per-device high-water mark (the latest timestamp already folded in) makes sure each flow is counted once. A baseline built on processing time would have learned when REEF was awake.

The tools, sorted by the job they do

Product names change faster than the jobs do. Fivetran and dbt Labs completed a merger on June 1, 2026, and IBM closed its acquisition of Confluent — the company built around Kafka — that March. Sorting by job survives that kind of churn. Table 3 does it that way.

JOB	WHAT IT DOES	EXAMPLES
Extract and load	Copies records from sources into a store, on a schedule or as changes happen	Fivetran, Airbyte, Debezium
Transform in the warehouse	Rebuilds cleaned and curated layers from raw, as versioned SQL	dbt
Store and query	Holds the raw and curated layers for analytical queries	Snowflake, BigQuery, Databricks
Publish and subscribe	Carries events from producers to any number of consumers	Apache Kafka, Google Cloud Pub/Sub, Amazon SNS and SQS, Amazon Kinesis Data Streams, Azure Event Hubs
Process streams	Computes over records as they arrive, with event time and watermarks	Apache Flink, Apache Beam on Google Cloud Dataflow, Spark Structured Streaming
Orchestrate	Schedules jobs and the dependencies between them	Apache Airflow, Dagster, Prefect
Test data quality	Asserts facts about data at the boundaries between layers	dbt data tests, Great Expectations, Soda
Serve features	Gives training and serving the same feature definitions	Feast

Table 3. Most of these tools overlap at the edges, and a pipeline needs something in each row whether or not it buys a product for it.

Two rows want a gloss. Change data capture, or CDC, reads a database's own transaction log and emits one event for every insert, update and delete. Debezium does this. It usually publishes the events to Kafka, which turns a database into a producer without touching the application that writes to it. Kinesis and Event Hubs

are partitioned logs in the Kafka mould, and Event Hubs will accept Kafka clients directly on its standard and higher tiers.

The rows that matter most for cleaning are the transform and the test. They share a habit worth copying with no tool at all. The cleaning rule is code. It lives in version control beside everything else, and every layer above raw can be rebuilt by running it again.

Every cleaning rule is a claim about the consumer

Data cleaning sounds like housekeeping. It behaves more like modelling, because every rule asserts something about what the consumer downstream needs. Dropping duplicate rows asserts that two identical records are one event. Filling a missing value with zero asserts that absence and nothing are the same thing. Is a record with no byte count a connection that moved nothing, or one the sensor failed to measure? The rule answers that question on the consumer's behalf, and it is only as good as its answer.

Start with the rules that need no learning. Most defects give way to them. A schema check refuses a record whose fields carry the wrong types, and a uniqueness test catches one key appearing twice. Range checks catch a negative byte count or a timestamp from next year. A referential check catches a device identifier that matches no known device. dbt ships four such checks as built-in data tests (unique, not null, accepted values and relationships); Great Expectations calls its version an expectation, a declarative assertion a pipeline can verify on every run.

The rule most pipelines get wrong is the one about absence. Absence has two causes. REEF has had to learn the difference more than once. When a feed can't be read, the honest result is a state that says so, and REEF marks that state as degraded. An empty list would tell every consumer downstream that the network had gone quiet — the one message a monitor can least afford to send by accident.

The same distinction once stood between REEF and the loss of its learned baseline. The file holding every device's profile was written in place, without an atomic replace. A reader could catch it half written, fail to parse it, and treat the failure as a

missing file. That reader would then start learning from scratch, flag every destination as novel, and save the empty profile over the real one. The defect surfaced during a refactor, before it had erased anything. The fix separated the two cases for good: a missing baseline starts fresh, and an unreadable one skips the pass and leaves the file alone until the next clean read.

Deduplication gets harder once two records disagree about how to name the thing they share. REEF's correlation pass fuses independent findings about one device and one destination into a single card with a confidence score. When it was written, it compared destinations as literal strings. That held while every detector named destinations by IP address. Later detectors named them by hostname, taken from the server name a device announces when it opens an encrypted connection, or from its DNS query. A finding about telemetry.example.com never fused with a finding about 203.0.113.7, even when the second address was what the first name had resolved to on that same device. The data was clean and spelled two ways.

Joining the spellings is entity resolution: deciding when records from different sources describe one real thing. Its formal literature goes back more than half a century, to the theory of record linkage that Ivan Fellegi and Alan Sunter published in the Journal of the American Statistical Association in 1969. The part REEF's fix got right is the part most matching code skips. An address that more than one name resolved to is ambiguous, so it keeps its own identity and merges with neither name, because merging unrelated findings would manufacture corroboration nobody supplied. Evidence that is missing or stale counts as unknown, which is a different answer from a mismatch. A false merge also ranks as worse than a missed one. The missed merge leaves two weak cards on a screen; the false one produces a single strong card that isn't true.

Where machine learning earns a place in cleaning

Rules stop scaling at two points. One is when the defects are too varied to list: a feed that drifts slowly, a vendor who changes what a field means without renaming it, a sensor that starts dropping one record in fifty. The other is when matching needs judgment, as it does with names, addresses and free text. Machine learning

comes into cleaning at both points. It brings a cost that rules don't carry. A learned cleaner can be wrong — confidently, and in ways nobody wrote down.

Anomaly detection on the data itself is the gentlest way in. A model learns what a normal batch looks like in row counts, null rates, value distributions and arrival times, and it flags a batch that departs from that. Isolation Forest is a common starting point. Fei Tony Liu, Kai Ming Ting and Zhi-Hua Zhou published it at the 2008 IEEE International Conference on Data Mining, and it isolates records by splitting the data at random; anomalies take fewer splits to isolate, because they sit apart from the crowd. Its output belongs on a batch as a flag. A detector that quietly drops whatever it finds strange is a filter nobody reviewed.

Learned matching is the second use. A classifier is trained on pairs of records that people have already labelled as same or different, and then it scores new pairs. It copes with the misspellings, abbreviations and reordered fields that break exact rules. The labels cost money. The thresholds are policy, and a pair with an uncertain score belongs in a review queue rather than in a merge.

Language models are the newest rung and the most tempting one. They can read a free-text field and pull out a structured value. They can collapse a vendor's ten spellings of one product into one, or propose a join between two schemas that nobody documented. They also produce fluent, plausible output when the right answer is "unknown" — exactly the failure a cleaning step can't afford. Used well, a language model proposes and something deterministic disposes: its output is validated against a schema, checked against the rules above it, and written to the cleaned layer with a record of where it came from.

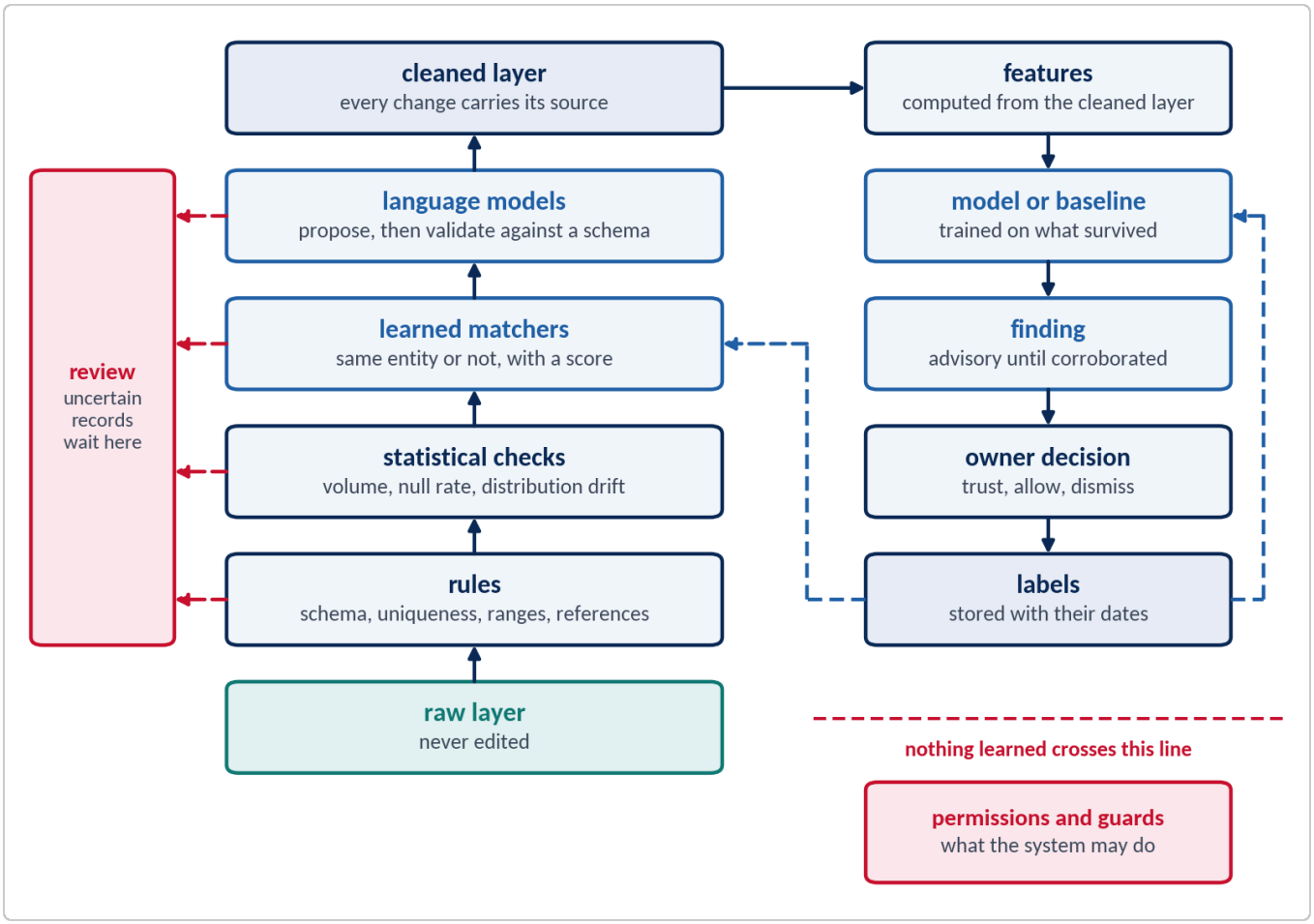


Figure 5. Deterministic rungs run before learned ones, every rung can send records to review, and nothing learned reaches the guards.

REEF's learning loop watches the firewall's settings change and asks for advice about them. It carries two rules worth lifting into any pipeline that lets a model touch data. A learned claim stays a claim: it's attributed, dated, and shown as something a model said rather than something the system observed. And learning can shape what REEF recommends while having no path to what REEF is permitted to do, since the guards that decide what may be blocked are not inputs to the loop and the loop is not an input to them. Figure 5 draws both rules into one picture of cleaning. The raw layer sits untouched beneath all of it.

Learning from what survived

Once the data is clean enough to trust, the last layer turns it into features. A feature is a numeric input a model actually reads. The CV of a device's connection gaps is one; so is the count of destinations it has ever contacted, and so are the hours of

the day when it's usually awake. How those features get computed matters as much as which ones get chosen.

The classic failure here is training-serving skew, a gap between the features a model saw in training and the ones it receives in production. Martin Zinkevich's Rules of Machine Learning, written at Google, gives a run of its rules to it. The usual cause is mundane. Training features came from a batch job over the warehouse; serving features come from a stream processor. The two implementations then disagree about an edge case, a time zone or a duplicate. A feature store such as Feast exists to close that gap: it keeps one set of feature definitions and serves them from an offline store for training and from a low-latency online store for prediction.

Time zones are the quiet version of the same fault. REEF's beacon detector reads any timestamp that carries no zone as UTC. It never reads one as local time, because a feed mixing marked and unmarked stamps would otherwise shift some gaps by hours and leave the rest alone. That rule rarely gets called cleaning. It decides whether a rhythm exists at all.

A behavioural model also has to decide when it knows enough to speak. REEF's baseline flags no drift for a device until it has seen that device on at least three separate days, and it caps a single run at twelve findings (so that one bad night can't flood the screen). A device with six or fewer known destinations counts as narrow-profile. Only those devices get flagged for reaching somewhere new. A laptop that contacts new servers every hour is exempt, since novelty is its normal.

The last decision is what a model's output is allowed to do. REEF's beacon findings start as advisory, on the reasoning that a heartbeat on its own is not malware. A beacon becomes actionable only when an authoritative reputation verdict agrees with it, or when at least two circumstantial signals do, such as a destination with no reverse DNS name together with a first-ever contact from a device that has an established history on the network. Corroboration is the gate. The loop that can take action automatically ships switched off. Each owner decision is also a label: a device marked trusted, a destination allowed, a finding dismissed as noise. A pipeline that stores those labels with their dates holds the training data any later model would need.

The heartbeat is still in the raw rows

Go back to the device keeping time at fifty-eight seconds. The merge that hid it was a cleaning decision made one layer too early — by code that had every reason to treat two copies of a connection as noise. Deduplication improved it. It didn't make it right. Keeping the feeds apart and letting the findings meet did, and the fix was cheap for one reason: nothing had learned from the merged table yet.

A model trained on that table would have learned something specific and wrong. It would have learned that this device's traffic is irregular. It would have learned that fifty-eight-second rhythms are rare here, and that a fraction of a second between two copies of every connection is a feature of normal life. Those are lessons about the table. They would sit in the weights and be applied quietly to every connection afterwards, and no test of the model would find them, because the model would be faithfully reproducing its data.

The raw rows would still show the heartbeat, one sensor at a time, fifty-eight seconds apart. Whether anyone could go back and read them depends on a decision taken before the first model was trained. Somebody decides where the raw rows live, and for how long.

Glossary

TERM	DEFINITION
At-least-once delivery	A broker's promise that every message arrives, at the cost that some arrive more than once. Most messaging systems default to it, so consumers have to tolerate duplicates.
Batch pipeline	A pipeline that runs on a schedule and processes everything that changed since its last run in one pass. It can't react faster than its schedule.
Beacon	A device calling one outside destination on a steady interval. Malware beacons to its command-and-control server, which makes a steady rhythm to an unknown destination worth a look.
Change data capture	Reading a database's transaction log and emitting one event per insert, update or delete, abbreviated CDC. It lets a database feed a stream with no change to the application writing to it.
Coefficient of variation	The standard deviation of a set of numbers divided by their mean, abbreviated CV. REEF applies it to the gaps between connections because it measures steadiness independently of tempo.
Command and control	The server an attacker uses to send instructions to compromised machines, shortened to C2. The contact is visible on the network even when its content is encrypted.
Consumer group	A set of consumers sharing one name that split a topic's partitions between them. Each partition is read by one member at a time; each group keeps its own position.
Deduplication	Collapsing records judged to describe one event into a single record. The key chosen decides the judgment, and a key that is too coarse or too fine changes what a downstream model learns.
Directed acyclic graph	A set of tasks joined by arrows that never loop back, abbreviated DAG. Orchestrators use one to know which job waits for which.

TERM	DEFINITION
ELT	Extract, load, transform: raw data lands in the destination first and is transformed inside it afterwards. Because the raw rows stay, a wrong cleaning rule can be fixed by rebuilding.
Entity resolution	Deciding when records from different sources refer to the same real thing, also called record linkage. Good matching treats an ambiguous case as unresolved, because a false merge invents agreement.
ETL	Extract, transform, load: data is reshaped on an intermediate server and only the result is stored. It removes data that must never persist, and it discards the raw version in the same step.
Event time	The moment a record's event actually happened. Processing time is the moment the pipeline handled it, and bucketing by event time keeps a late record in the period it belongs to.
Exactly-once processing	A guarantee that each record affects the result once, across retries and failures. It is always scoped to a particular path, such as reading from and writing back to one Kafka cluster.
Feature	A numeric input a model reads, computed from cleaned data.
Feature store	A system that holds feature definitions and serves them both to training jobs and to live predictions. It exists so that training and serving never compute the same feature two ways.
Idempotent	Describes an operation that changes nothing when repeated with the same input. An idempotent consumer can safely receive a message twice.
Isolation Forest	An anomaly detection method, published by Liu, Ting and Zhou in 2008, that isolates records with random splits. Anomalies need fewer splits, so it can score unusual batches without labelled examples.
Medallion architecture	A layering of raw, cleaned and curated data, named bronze, silver and gold in Databricks documentation. Every layer above raw can be rebuilt from the one below.

TERM	DEFINITION
Offset	A consumer's position in one partition of a log-based topic, stored as a number. Rewinding it replays history.
Orchestrator	A scheduler that runs pipeline jobs in dependency order and retries or alerts when one fails. Airflow, Dagster and Prefect are orchestrators.
Partition	One ordered slice of a topic in a log-based system such as Kafka. Order holds within a partition and not across partitions, so records that must stay in order share a key.
Publish and subscribe	A messaging pattern in which producers send messages to a named topic and any number of subscribers receive them, shortened to pub/sub. Neither side needs to know the other exists.
Quarantine	A holding area for records that failed a quality check. It turns a failed check into a queue a person can work through; a silent drop leaves nothing to review.
Streaming pipeline	A pipeline that processes records as they arrive. Anything that has to react quickly, a detector included, runs on one.
Topic	A named channel in a pub/sub system that producers write to and subscribers read from.
Training-serving skew	A difference between the features a model was trained on and the features it receives in production. It degrades a model silently, because each feature on its own still looks plausible.
Warehouse	A database built for analytical queries over large volumes of data. Under ELT it holds the raw data as well as the cleaned layers built from it.
Watermark	A stream processor's running declaration that event time has reached a given point, so earlier records are no longer expected. It decides when a time window can close, and what happens to a record that arrives after it.

Newell, J. (2026). MAX Research Collective, WPR-DAT-007. CC BY 4.0

DATA WPR-DAT-007

// ABOUT THE AUTHOR

Jason Newell leads organizations and people in technology, and is a researcher, deep technologist and builder — from a CRAY to agentic AI. He founded MAX Research Collective to build AI for regulated industries that runs where its owner has control, with healthcare as its strongest focus. About Jason → Connect on LinkedIn →