LOGNROLL

FREE TECHNICAL BOOK · FIRST EDITION · SEPTEMBER 2026

How Session Replay Service Works

Inside the Architecture of a Session Replay Platform — from Recording to Replay

Written by the LogNroll Team

HTML · PDF · EPUB · lognroll.com/book

Chapter 21: Epilogue — and Should You Build or Buy?

About this book

Somewhere on your website, a visitor is stuck. They clicked a button and nothing happened. They filled in a form, pressed Pay, and were charged — twice. They are not going to email you a detailed bug report with repro steps. They are going to close the tab and go to a competitor, and the only record of what went wrong will be… nothing. Unless you recorded the session.

This book is about the software that records those sessions. Session replay services — tools in the family of LogRocket, FullStory, Hotjar, and the platform this book is written around, **LogNroll** — let you watch a user's journey through your product as if you were standing behind them: every click, scroll, typed character, console error, and network request, reconstructed on your screen after the fact.

The surprising part is that no video is ever recorded. A session replay service captures *events* — small, structured records of what happened — and later *reconstructs* the page from them. That one design decision drives everything else in this book: the wire format, the batching, the message bus, the storage tiers, the processing jobs, and the player that puts time back together. It is also why session replay is practical at all: a text event stream is a hundred times smaller than a video of the same session, it can be searched, it can be analyzed, and — done right — it can respect the privacy of the people using your site.

We wrote it for engineers, engineering managers, founders, and curious support leads. You will come away understanding how these systems actually work, what each moving part is for, where the costs hide, and how to make good decisions — whether you are buying a session replay service or building one.

The story inside this book

Technical books can be dry. So woven through these pages is a story: **Candlewood Books**, a small independent online bookstore, and the spring when everything went wrong. Orders charged twice. Buttons that did nothing. A furious librarian client threatening to leave. A support lead who could not reproduce a single bug, and a founder watching her shop's reputation drain away one bad review at a time.

> **A note to the reader:** Candlewood Books, its people, its clients, and every number in its story are fictional — a composite drawn from the thousands of small companies we have seen struggle with the same problems. The product problems are real; the names are invented. Where the story reports numbers like "tickets dropped by 70 percent," treat them as story numbers, not statistics.

Candlewood's arc runs through the book like a spine. Each technical chapter ends with a short "Story checkpoint" that shows how the machinery you just learned about changed what that little company could see — and, in the end, whether it survived.

Who this book is for

- **Frontend and full-stack engineers** who use session replay at work and want to know what happens between the recorder and the replay.
- **Engineers who are considering building** their own replay pipeline (on top of libraries like rrweb) and need an honest map of the work — Chapters 11–16 and 21 are for you.
- **Engineering managers and founders** who want to understand the cost structure, privacy tradeoffs, and operational realities of recording user sessions — Chapters 7, 12, 18, and 19.
- **Product and support people** who live in replays every day and want vocabulary and mental models to talk to engineers on equal footing.

No prior knowledge of session replay is assumed. We do assume you are comfortable reading code: there are examples in TypeScript, Go, Java, protobuf, and JSON, but the ideas carry the text.

How this book is organized

The book follows the journey of a single session from the moment a page loads to the moment a support agent watches it back, in six parts:

- **Part I — The Why and the World.** What session replay is, why it exists, the story of Candlewood Books, and the anatomy of a session (Chapter 1–3).
- **Part II — Recording.** The recorder SDK, how every kind of browser event is captured, the protobuf wire language, privacy and masking, and how data gets out of the browser (Chapter 4–8).
- **Part III — Ingestion, Bus, and Storage.** The receiver gateway, the NATS JetStream message bus, the archiving worker, and the storage lifecycle from ACTIVE to REMOVED (Chapter 9–12).
- **Part IV — Making Sessions Useful.** The session processor that turns raw events into errors, heatmaps, and insights, and the background jobs and alerts that keep a replay platform (and the sites it watches) healthy (Chapter 13–14).
- **Part V — Replay and the Product.** The player API and the player itself — how a browser rebuilds a page and replays time — plus the analytics product around replay (Chapter 15–17).

- **Part VI — Running It for Real.** Operating a session replay service at scale, the performance and cost math that determines whether the business model works, the future of the technology, and the closing story of Candlewood Books (Chapter 18–21).

Five appendices collect the reference material: the full protobuf contract (A), the storage and collection reference (B), a configuration reference (C), a glossary (D), and further reading (E).

Conventions used in this book

- Code and identifiers appear in `monospace`. Payload field names, status strings, and endpoints match the reference implementation exactly.
- Callout boxes appear as blockquotes with a bold label:

> **Field note:** Something worth remembering.

- ASCII diagrams in monospace blocks show architecture and data flow. They are deliberately simple: enough to carry the idea, not to replace the code.
- Terms are **bolded** on first use.
- Numbers are written out ("nine sessions") or as numerals ("10,000 sessions") using the convention that one through nine are spelled and 10 and up are digits.
- All times are relative to the event stream's `timestamp`, which is milliseconds since the Unix epoch, unless stated otherwise.

The reference platform: LogNroll

The book explains session replay services in general, but every mechanism is grounded in one concrete, real system: **LogNroll**, a session replay and product analytics platform built as a set of small services — a browser recorder, an ingestion gateway, a message bus, an archiving worker, a processor, a replay API, and an Angular player. We chose it because it is a real, running implementation: the constants you will meet (batch sizes, timeouts, retention windows, endpoints) are the constants in its source code, not idealized values. Where we describe something that is specific to LogNroll rather than universal to the category, we say so.

Two honest caveats. First, the LogNroll codebase is a living thing — by the time you read this, some details will have moved on; the ideas will not. Second, we wrote it with the platform's future in mind: Chapter 20 is explicitly a roadmap of improvements we believe the architecture (and the industry) is heading toward, including work that was already underway when this book went to press.

Free, forever

This book is free. Read it online, download the PDF or EPUB, keep a copy on your team's shelf, send it to a colleague who is debugging the same checkout bug you are. It is released under the **Creative Commons Attribution 4.0 (CC BY 4.0)** license: you may share and adapt it for any purpose, even commercially, as long as you give credit to the LogNroll team. You can find the license text at `https://creativecommons.org/licenses/by/4.0/`.

Why give away two hundred pages of hard-won engineering knowledge? Because session replay only gets better when more teams understand it — when buyers know what to ask for, when builders know what they are signing up for, and when the people whose sessions are recorded are treated with the respect the technology owes them. We would rather be the reference people learn from than the tool they blame.

First edition, September 2026. Published by the LogNroll team. Written with care, from source code and from the field.

Acknowledgments

Thanks to the engineers who built the platform this book documents, and to every support agent who has ever said "can you show me what happened on your screen?" — this book is for you. And to the fictional Maya Okafor: we made you up, but we have met you a hundred times, and you are the reason this book exists.

— The LogNroll Team, September 2026

Appendix A. The LogPoint Contract

Every event a session replay recorder produces is wrapped in one protobuf message. This appendix documents the contract in full, as it exists in the reference implementation.

A.1 The schema

```
syntax = "proto3";

option java_package = "com.lognroll.receiver";
option java_outer_classname = "LogPointProto";

message LogPoint {
  int64 timestamp = 1;
  LogType type = 2;
  bytes data = 3;
  fixed32 version = 4;
  int64 order = 5;
  int64 index = 6;

  enum LogType {
    NAVIGATION = 0;
    MUTATION = 1;
    CLICK = 2;
    INPUT = 3;
    MOUSE_MOVE = 4;
    LOG = 5;
    NETWORK = 6;
    SCROLL = 7;
    KEYBOARD = 8;
    FORM = 9;
    META = 10;
    IDENTIFY = 11;
    STYLES = 12;
    PING = 13;
    PERFORMANCE = 14;
  }
}

message LogPoints {
  repeated LogPoint items = 1;
}
```

A.2 Field reference

Field	Type	Meaning
`timestamp`	int64	Milliseconds since the Unix epoch when the event happened in the browser. The replay timeline is built from this value.

Field	Type	Meaning
`type`	LogType	Which kind of event this is (see A.3). The player and the processor dispatch on this value.
`data`	bytes	The type-specific payload. For most types it is a small JSON document; because it is `bytes`, payload formats can evolve without changing the envelope.
`version`	fixed32	Version of the payload/processing semantics for this event type. Consumers use it to interpret `data` correctly (see A.4).
`order`	int64	Sequence number carried by DOM-mutation events, preserving the browser's delivery order for mutations — the event class where application order affects replay correctness.
`index`	int64	Per-event counter stamped by the recorder and persisted across page reloads; the chronological tiebreaker when timestamps collide.

A.3 Event types

Value	Name	What it carries (typical payload contents)
0	`NAVIGATION`	A page navigation or SPA route change: URL, referrer, load timing.
1	`MUTATION`	A DOM change: nodes added or removed, attribute or text changes — the raw material for rebuilding the page during replay.
2	`CLICK`	A mouse click: coordinates, element tag, per-element offsets, and an XPath that identifies the element.

Value	Name	What it carries (typical payload contents)
3	`INPUT`	A text input or value change in a form control.
4	`MOUSE_MOVE`	A mouse position sample; heavily throttled and batched.
5	`LOG`	A console message: `log`, `warn`, `error`, with the message text and stack where available.
6	`NETWORK`	A captured network request: method, URL, status, duration, and body summary.
7	`SCROLL`	A scroll position change, absolute and element-relative.
8	`KEYBOARD`	A key press, usually captured as part of form entry context.
9	`FORM`	A form submission or form-field event.
10	`META`	Page metadata: title, language, viewport size.
11	`IDENTIFY`	Identity information set by the site owner, such as a user email or ID, so sessions can be linked to a person.
12	`STYLES`	Style-sheet information the player needs to render the page as the user saw it.
13	`PING`	A heartbeat sent while the session is alive but quiet, so the platform can tell an open tab from a dead one.
14	`PERFORMANCE`	Browser performance measurements (timings, Core Web Vitals style metrics).

A.4 Versioning and the “data as bytes” rule

The envelope’s `version` field exists because payload formats evolve. A player that understands version 1 of a `NETWORK` payload should still be able to skip (or best-effort render) a version 2 payload it has not seen yet. The rule the platform follows: **bump the version whenever the shape or meaning of a payload changes**, and make consumers gate on the version rather than assume.

The same discipline appears server-side in the session processor: each processor records the version it has applied to a session, and the pipeline skips reprocessing sessions whose applied version is already current. Treat this as a general law of replay systems: events are written once and read many times, often by software written years later — the format is a contract with your future self.

A.5 One contract, many copies

In the reference implementation, the `.proto` file is duplicated across the logger, receiver, worker, processor, and player repositories, and the player additionally keeps a generated TypeScript module. This is a real and acknowledged maintenance risk: a new event type must be added in several places in lockstep. The future direction (Chapter 20) is to publish the contract as a shared package with a single version. For a small team this duplication is survivable; the lesson for anyone building a replay system is to decide early whether the contract lives in one shared artifact or in N copies — and to know which one you chose.

Appendix B. Storage and Collections Reference

A session replay platform is, at heart, a database problem wearing an event-stream costume. This appendix documents where the reference implementation stores what: the coordination store (MongoDB), the content store (S3-compatible object storage), and the derived collections produced by processing.

B.1 The `sessions` collection (MongoDB)

Every service that touches a session — receiver, worker, processor, status job, remover job, player API, main API — reads or writes the same shared collection. The document below is a representative view of the fields in play across those services (field sets differ slightly between services; the core is stable).

```
{
  "_id": "65f1a2b3c4d5e6f7a8b9c0d1",
  "companyId": "64e0f1a2b3c4d5e6f7a8b9c0",
  "userId": "auth0|64e0...",
  "userName": "helena@example.com",
  "userEmail": "helena@example.com",
  "deviceId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "startTime": "2026-05-02T14:03:11.200Z",
  "endTime": "2026-05-02T14:21:47.903Z",
  "lastInteractionTime": "2026-05-02T14:21:40.110Z",
  "lastUpdateTime": "2026-05-02T14:21:47.903Z",
  "duration": 1116643,
  "urls": [
    "https://shop.candlewood.example/",
    "https://shop.candlewood.example/books/mystery",
    "https://shop.candlewood.example/checkout"
  ],
  "browser": "Chrome",
  "os": "macOS",
  "ip": "203.0.113.24",
  "country": "DE",
  "city": "Berlin",
  "status": "FINISHED",
  "storageName": "SPACES",
  "s3ServiceName": "s3SpacesService",
  "lockedBy": "worker-7f9c…",
  "lockTimestamp": "2026-05-02T14:22:02.000Z",
  "events": ["NETWORK", "MUTATION", "CLICK"],
  "numberOfInteractions": 214,
  "processedVersion": 4,
  "processedTimestamp": 1752438231000,
  "processedProcessors": { "errorDetection": 3, "clickHeatmap": 4, "scrollHeat": 2 },
  "retries": 0,
  "crmCounted": true
}
```

Key points:

- `companyId` is the multi-tenant boundary: every query in the product is scoped to it, and the player API authorizes reads by checking the caller's membership in the session's company.
- `status` drives everything. The lifecycle is `ACTIVE → IDLE → FINISHED → REMOVED`, with `FAILED` as an exit path used when a session can never be archived. Transitions are made by different services: the receiver marks a session active and refreshes last-seen times — and nudges sessions without interaction toward IDLE; the status job moves stale sessions to FINISHED; the worker archives FINISHED sessions; the remover job deletes FINISHED sessions older than 30 days.
- `lockedBy` / `lockTimestamp` implement a crude distributed lock: a worker or job claims a session with an atomic conditional update, and other claimants skip it while the lock is fresh. Locks expire (roughly two minutes for the archiving worker, about ten minutes for the remover job) so a crashed claimant cannot strand a session forever.
- `storageName` / `s3ServiceName` record where the session's archive lives and which storage tenant was chosen for it (see B.3). The worker only claims sessions that have not yet been archived.
- `processedProcessors` maps processor name to applied version. This is the mechanism that makes reprocessing safe: a processor whose version is already applied to a session is skipped (see Chapter 13).

B.2 Derived collections

The session processor reads archived sessions and writes derived collections that power the product UI. These are the real collection names used by the platform:

Collection	Written by	Purpose	Notable fields
`sessionErrors`	error-detection processor	Console errors and uncaught exceptions per session	message, stack, source, timestamp, severity
`backendRequests`	network-analysis processor	Network requests worth surfacing (failed, slow, or noteworthy)	method, url, status, durationMs, request/response summary
`heatMapClicks`	click-heatmap processor	Click positions aggregated across sessions	xpath (the stable cross-session identity), relX, relY, deviceType, session/url context

Collection	Written by	Purpose	Notable fields
`scrollHeat`	scroll-heat processor (v2)	How far users scroll and how long they dwell per 10% band of the page	dwellMs[], maxDepth, totalDwellMs, viewportHeight, deviceType, version
`hostMonitors`	host-monitoring module	Uptime/TLS watchlist per company origin	origin, source (DISCOVERED/MANUAL), status (UP/DOWN/UNKNOWN/PAUSED), cert state, alertEnabled
`hostUptimeSamples`	host-monitoring module	Probe history per host (30-day TTL)	ok, responseMs, checkedAt
`hostAlerts`	host-monitoring module	Alert history, one per incident	kind (DOWN/RECOVERY/CERT), firedAt, clearedAt

Two rules that keep these collections honest, both inherited from the processor design:

1. **Cross-session identity must be stable.** For click heatmaps the identity is the element's XPath, which is the same on every visit. Per-session counters like `lnrId` must never be used to merge clicks across sessions.
2. **Shape changes require a version bump.** When a processor's output semantics change, its `PROCESS_VERSION` is bumped, the session's `processedProcessors` entry gates reprocessing, and readers filter on the version they understand (for example, scroll-heat readers require `version >= 2`).

B.3 The archive (S3 / DigitalOcean Spaces)

The content store holds, per session, a set of zipped protobuf chunk files. The layout is `sessions/{sessionId}/…`, where each object is a zip containing one or more binary `*.logpoints.pb` frames (the protobuf `LogPoints` batch format). Chunking is deliberate: the player can fetch the frames it needs instead of the whole session, and no single S3 object grows without bound.

Two storage tenants exist, chosen per session (recorded in `s3ServiceName`):

Tenant	Region / bucket	Used for
`lognroll-test` (fra1)	DigitalOcean Spaces, Frankfurt	Sessions of the primary tenant

Tenant	Region / bucket	Used for
`sessionreplay` (ams3)	DigitalOcean Spaces, Amsterdam	Sessions of the second tenant (POLISUA)

B.4 Retention and cleanup

- **Receiver**: marks a session `ACTIVE` on first contact, refreshes it while batches arrive, and nudges a session without user interaction toward `IDLE` (a softer state that the product treats as "still alive but quiet").
- **Status job**: runs on a schedule (every minute by default) and moves stale sessions to `FINISHED`. Its thresholds are short (a session is considered dead after a couple of minutes of silence from its batches, longer without real user interaction, with a ceiling on total session length). FINISHED is the signal to the worker that a session is complete and should be archived.
- **Worker**: claims FINISHED, un-archived sessions, archives them to S3, purges their message-bus subjects, and releases the lock.
- **Remover job**: runs frequently in production (every few minutes; hourly in development) and claims FINISHED sessions whose start time is older than the retention window (the cutoff is computed as thirty-one 24-hour periods from start time — the platform advertises the rounder "30 days"), deletes their S3 objects, and sets `status = REMOVED`. From the product's point of view a REMOVED session is gone: the player API returns nothing for it, honoring the retention promise made to customers (and, by extension, to the people whose sessions are recorded).

The 30-day window is a product decision with an engineering shape: it bounds storage cost, bounds privacy exposure, and — because the archive is cold and cheap — it can be extended per plan without changing any of the machinery above.

Appendix C. Configuration Reference

Every service in the reference implementation is configured with environment variables read at startup, with sensible local-development defaults baked in. Secrets (encryption keys, S3 credentials, JWT secret, NATS password) come from a Kubernetes secret in production, never from defaults. Where the tables below show a default of "(none)", the source code contains a non-functional stand-in string that is always overridden by the deployment environment — treat such values as required configuration.

The tables below list the variables and their defaults exactly as they appear in the services' config code. Defaults differ between services and environments on purpose (for example, the receiver's NATS subject default is `sessions3-dev`, matching the dev stream) — treat each service's own configuration as authoritative for that service.

C.1 Receiver (ingestion gateway)

Variable	Default (dev)	Purpose
`APP_ENV`	`dev`	Deployment environment tag
`SERVER_PORT`	`8383`	HTTP listen port
`MONGODB_URI`	`mongodb://localhost:27017/lognroll-dev`	MongoDB connection string (database name comes from the URI path)
`NATS_URL`	`nats://nats.nats-system.svc.cluster.local:4222`	NATS server
`NATS_USER` / `NATS_PASSWORD`	`admin` / `pass`	NATS credentials
`NATS_STREAM`	`sessions3-dev-stream`	JetStream stream name
`NATS_STREAM_SUBJECT`	`sessions3-dev`	Subject prefix; a session's subject is `{prefix}.{sessionId}`
`ENCRYPTION_KEY`	(none)	AES key used to encrypt point payloads before publishing
`IP_API_KEY` / `IP_API_HOST`	key / `https://pro.ip-api.com`	IP geolocation provider
`REAL_IP_HEADER`	`CF-Connecting-IP`	Header that carries the real client IP behind Cloudflare

C.2 Worker (archiver)

Variable	Default (dev)	Purpose
`HTTP_PORT` / `ACTUATOR_PORT`	`8484` / `8181`	App and health-probe ports
`MONGODB_URI`	—	MongoDB connection string
`ENCRYPTION_KEY`	(none)	AES key to decrypt points
`S3_ENDPOINT` / `S3_REGION` / `S3_BUCKET`	`https://fra1.digitaloceanspaces.com` / `fra1` / …	Primary archive tenant (lognroll-test)
`S3_ENDPOINT_POLISUA` / `S3_REGION_POLISUA` / `S3_BUCKET_POLISUA`	`https://ams3.digitaloceanspaces.com` / `ams3` / …	Second archive tenant (POLISUA)
`NATS_URL`, `NATS_USER`, `NATS_PASSWORD`, `NATS_STREAM_SUBJECT`	…	NATS consumer configuration
`IP_API_KEY` / `IP_API_HOST`	…	Geolocation enrichment

C.3 Player API

Variable	Default (dev)	Purpose
`HTTP_PORT` / `ACTUATOR_PORT`	`8080` / `8181`	App and health-probe ports
`MONGODB_URI`	—	MongoDB connection string
`ENCRYPTION_KEY`	(none)	AES key to decrypt archived points
`S3_ENDPOINT`, `S3_REGION`, `S3_ACCESS_KEY`, `S3_SECRET_KEY`, `S3_BUCKET` (+ `_POLISUA` variants)	…	Archive tenants (same shape as the worker)
`JWT_SECRET`	(none)	Secret used to validate app-issued JWTs
`CONTEXT_PATH`	`/api`	URL prefix for API routes

C.4 Lifecycle jobs

Variable	Default (dev)	Purpose
`MONGODB_URI`	`mongodb://localhost:27017/lognroll-dev`	MongoDB connection string
`S3_ENDPOINT`, `S3_REGION`, `S3_ACCESS_KEY`, `S3_SECRET_KEY`, `S3_BUCKET` (+ `_POLISUA` variants)	…	Archive tenants (remover job only — it deletes objects)

C.5 A note on the “two tenants” pattern

Several services accept two complete sets of S3 configuration, suffixed `_POLISUA`. A session’s `s3ServiceName` selects which tenant archives it. This is how one platform can host two tenants with separate buckets and regions from a single deployment — and it is a pattern worth understanding before you build multi-region or multi-tenant storage of your own.

C.6 General rules

- **Never ship real secrets as defaults.** Every default credential above is either a local convenience value or a non-functional stand-in; production values arrive via environment or Kubernetes secrets.
- **Database names come from the URI.** Several services derive their database name from the path in `MONGODB_URI` (`lognroll-dev` vs `lognroll-prod`) rather than hard-coding it — a small convention that prevents a service from reading the wrong database in the wrong environment.
- **Two ports, two jobs.** Services expose an application port and an actuator/health port so that Kubernetes liveness and readiness probes never contend with application traffic.

Appendix D. Glossary

Term	Definition
ACK	Acknowledgement of a message on a queue; the consumer tells the broker it has finished processing a message so it can be removed.
Archive	The long-term, cold storage of a session's event stream (in the reference platform: S3-compatible object storage).
At-least-once delivery	A delivery guarantee in which a message may be redelivered, so consumers must be idempotent. Cheaper than exactly-once.
Batch	A group of events collected on the client and sent in one request to reduce overhead.
Beacon / sendBeacon	A browser API that lets a page send small amounts of data reliably even while the page is being unloaded.
Chunk	A segment of an archived session (a zipped protobuf frame). Chunking lets the player fetch only what it needs.
Claim (lock)	The act of atomically marking a session as being processed by one worker so no other worker picks it up.
Cold storage	Slow but very cheap storage for data that is rarely read (finished sessions).
Company	The multi-tenant boundary in the reference platform; a customer account that owns sessions and users.
Console capture	Wrapping `console.log/warn/error` so messages become replayable events.
CORS	Cross-Origin Resource Sharing; the browser mechanism that lets a recorder on one origin send requests to a receiver on another.
Data minimization	The principle of collecting only the data you actually need; a core privacy practice for replay.
DOM	The Document Object Model — the browser's structured representation of a page; mutations to it are the raw material of replay.

Term	Definition
DOM mutation	Any change to the page tree: nodes added or removed, attributes or text changed.
Event stream	The ordered sequence of events that constitutes a recorded session.
Exactly-once delivery	The strongest (and most expensive) delivery guarantee; rarely needed when downstream processing is idempotent.
Heatmap	An aggregated visual of where users clicked or how far they scrolled, built from many sessions.
Idempotent	Safe to apply twice; a property downstream processors and jobs rely on when messages are redelivered.
Identify	Associating an anonymous session with a known user (email or ID) so replays can be searched and attributed.
Ingestion gateway (receiver)	The first server a recorder talks to; validates and stores events, then hands them to the pipeline.
JetStream	NATS' built-in persistence layer that gives streams, retention, and replay of messages.
JWT	JSON Web Token; the signed token used to authenticate API requests between the product's services and UIs.
Lock timestamp	The time a worker claimed a session; used to expire locks of crashed workers.
Log point	The platform's name for a single recorded event (protobuf message with timestamp, type, data, version, order, index).
Masking	Preventing sensitive input values (passwords, card numbers) from being recorded or from being replayed in clear text.
Message bus	A durable channel decoupling producers from consumers (here: NATS JetStream).
Multi-tenancy	One platform serving many customer accounts with strict data isolation between them.

Term	Definition
MutationObserver	A browser API that reports DOM changes asynchronously; the workhorse of DOM recording.
NATS	A lightweight message broker; with JetStream it provides persistence and consumer groups.
Order / index	Event sequence fields that let consumers reconstruct exact arrival order.
Player	The frontend that reconstructs and replays a session from its event stream.
Processor	A backend job that reads an archived session and derives structured insights (errors, heatmap clicks, scroll heat, network analysis).
Protobuf (Protocol Buffers)	Google's compact, schema-driven binary serialization format.
Rage click	Rapid repeated clicking in one spot — a strong signal of frustration, often on a broken element.
Receiver	See *Ingestion gateway*.
Recorder (logger)	The browser-side JavaScript that captures events.
Retention	How long session data is kept before deletion (here: 30 days by default, then REMOVED).
rrweb	An open-source library for recording and replaying DOM events; a common starting point for building custom replay.
Sanitizer	Client-side logic that strips sensitive material (headers, tokens, secrets) from captured network payloads.
Scroll heat	Analysis of how far users scroll and how long they dwell in each band of a page.
SDK	The installable package site owners embed (here: `@lognroll/lib`) that initializes the recorder.
Session	One continuous visit by one user on one device, from first event to timeout or close.
Session id	The unique identifier of a session; created at first contact (NEW) and reused for the session's lifetime.

Term	Definition
Snippet	A few lines of script a site adds to load the recorder without a package manager.
Subject	A NATS topic name; the platform uses one subject per session to preserve ordering.
TTL (time-to-live)	Automatic expiry of data after a set period (e.g., probe samples kept 30 days).
Web Worker	A browser thread separate from the main page thread; used here for batching and transport so the UI never blocks.
XPath	A path expression identifying an element in the document; used as the stable cross-session identity for click heatmaps.
XHR	XMLHttpRequest; together with `fetch`, the browser API for network requests, both wrapped by recorders to capture traffic.

A few numbers worth remembering

- 15 event types in the `LogType` enum (0–14).
- 200 events per client batch, flushed every 100 ms (mouse moves every 200 ms).
- 2 minutes: worker lock expiry.
- 30 days: default retention window before a FINISHED session is REMOVED.
- 5 minutes: default uptime-probe interval; 24 h/7 d/30 d: uptime reporting windows.
- 2 storage tenants (fra1 and ams3), 2 ports per service (app + actuator), 1 shared encryption key (soon to be many, see Chapter 20).

Appendix E. Further Reading

From the LogNroll blog

The LogNroll engineering blog (lognroll.com/blog) covers session replay from angles this book touches only briefly — all free to read:

- *Session Replay Architecture* — a tour of the platform from the recording script to the player.
- *Core Web Vitals and Session Replay UX* — how replay connects to performance monitoring.
- *Error Tracking + Session Replay: Production Debugging* — the debugging ladder from Chapter 1.
- *Heatmaps vs Session Replay: When Each Wins* — when aggregated views beat individual replays (relevant to Chapters 13 and 17).
- *Rage Clicks to Roadmap* — turning frustration signals into product decisions.
- *Network Debugging with Session Replay* — the network timeline (Chapter 16) in action.
- *Debug Frontend Bugs with Replay* — a practical, incident-led guide.
- *Privacy-First Session Recording in the EU* — GDPR considerations (Chapters 7 and 18).
- *Local LLMs for Session Replay Debugging (Privacy)* and *MCP Server: Session Data for AI Agents* — the future of AI-assisted session analysis (Chapter 20).
- *Automatic Host Monitoring: Uptime & TLS Alerts* — the monitoring module (Chapter 14).

Platform documentation

- LogNroll docs — quickstart and configuration: lognroll.com/docs
- LogNroll privacy configuration: lognroll.com/docs/configuration/privacy

Standards, tools, and open source

- **Protocol Buffers** — protobuf.dev (schema language, encoding, language guides). The wire-format details in Chapter 6 are documented there.
- **NATS and JetStream** — docs.nats.io (streams, subjects, consumers, exactly-once vs at-least-once semantics).
- **rrweb** — github.com/rrweb-io/rrweb — the open-source DOM recording/replay library referenced in Chapters 16 and 21.

- **MDN Web Docs** — the authoritative references for the browser APIs a recorder leans on: MutationObserver, Web Workers, sendBeacon, PerformanceObserver, and CSP.
- **GDPR** — the full regulation text: gdpr-info.eu. For practical guidance on recording sessions lawfully in the EU, see the ICO's guidance on analytics and the ePrivacy Directive's consent requirements.

Licensing

Chapter 1: What Is a Session Replay Service — and Why It Exists

It happens in some software company every single week. A team ships a new version of its website on Monday night. The deploy is green. The tests pass. The developer who wrote the checkout goes home satisfied. On Tuesday morning the first ticket arrives: “I tried to buy a book and the button just sat there. Nothing happened.” By Wednesday there are 11 tickets saying the same thing in 11 different ways: the button did nothing, the page froze, the site charged me twice, the confirmation email never came. The developer opens the checkout on her own machine. It works. She tries another browser. It works. She calls a colleague. It works for him too. “It’s probably the customer’s browser,” someone says. “Ask them to clear their cache.” And somewhere out there, a real person who wanted to give the company money walks away and never comes back.

That scene is so common because of a structural fact: the people who build and run a website cannot see what the people who use it actually see. A server can log every request it receives. A database can record every order. But between the moment a page arrives in a visitor’s browser and the moment that visitor gives up, the product runs on hardware you do not own, in a browser you cannot open, in front of a person you cannot watch. For most of the history of the web, that stretch was simply dark.

Session replay is the technology that switches the light on. It is a category of tooling — in the same family as LogRocket, FullStory, and Hotjar replays — that records what happens inside a real user’s browser session and lets you watch it later as if you were standing behind that user’s chair: where the mouse went, what was clicked, what was typed, what appeared on the page, and what went wrong. This book explains how such services work, using one real platform, LogNroll, as a walking-through reference. And the first thing to understand is a trick the name hides: session replay is not what most people picture.

What session replay is

When people first hear “session replay,” many imagine a screen recording: a video file of the user’s screen, uploaded and streamed back later. That is not how the services in this book work — and the difference is not an implementation detail. It is the whole point.

A session replay service does not record pixels. It records **events**: small, structured, timestamped descriptions of what happened in the page. The page loaded. The user scrolled to the reviews. The mouse paused over the “Add to cart” button and clicked. The cart opened. The user typed into a field. A new section of the page appeared without a reload. The console printed an error. Each moment becomes one record in a timeline, carrying enough

information to rebuild the page later: what it looked like, what changed, and what the user did against it.

To replay a session, a player application reads that timeline and reconstructs it, rebuilding the page the user saw and re-enacting the cursor, the scrolls, the clicks, and the typing in order. The result looks like a video and for most purposes feels like one, but it is not. Think of the difference between a film of a stage play and the play's script with its stage directions: given the script and the blocking, you can perform the play again any time, at any speed — and search it for the exact moment someone dropped a line.

	Screen video	**Event-driven replay**
What is stored	Pixels, frame after frame	Structured events with timestamps
Size per minute	Megabytes, typically	Orders of magnitude less
Privacy control	Hard: a pixel is a pixel	You choose what to record; sensitive data can stay out of the stream entirely
Search	You cannot search a video	Events are data — queryable, filterable, countable
Beyond watching	Nothing extra	Feeds analytics: errors, heatmaps, performance, funnels

Why does reconstruction win? Start with size. An event is a few coordinates or a few fields, so a whole session costs a sliver of what video would — which matters when you store many sessions a day. Privacy is the deeper win, and we will return to it, but the short version is this: a video cannot un-see a credit card number that happened to be on screen, while an event stream can simply decline to record it in the first place. And because events are structured data rather than pixels, they can be searched, filtered, aggregated, and counted: find every session where checkout errored, or every click on one button.

Honesty matters here too: a reconstruction is not a photograph. Canvas animations, video players, and cross-origin iframes are genuinely hard to reproduce faithfully, and later chapters discuss those limits openly. For most real websites — forms, catalogs, dashboards, checkout flows — the reconstruction is faithful enough that you can watch a stranger's afternoon and understand exactly what happened.

Which brings us to the word "session," which has a precise meaning in this world. A session is one continuous visit by one person on one device: it begins when recording starts, ends when the person leaves or goes idle long enough, and it carries one identity throughout — the

session id every recorded event is tagged with. Everything in this book hangs off that idea: a session is the complete, ordered story of one visit, kept as data rather than as film.

Why the blind spot exists

The “works on my machine” problem is older than the web, but the modern web made it worse. Two decades ago most of a site’s behavior lived on the server, where the logs could see it. Today the center of gravity has moved into the browser: JavaScript builds the page, validates the form, talks to APIs, updates the screen. If the logic that decides whether the “Pay now” button works lives in a browser on a customer’s phone, then the one place you cannot look is the one place the bug lives.

Your existing tools cannot see it either. Logs tell you what your servers did; they are silent about what the visitor’s browser did or failed to do. Metrics tell you that checkout completion fell by a third last month, but not why. A database tells you an order never arrived; it cannot tell you that the customer pressed the button four times and watched it do nothing. Support is left to interrogate the invisible — which browser, which phone, can you send a screenshot — and screenshots do not move. “Please clear your cache” became a punchline because it was the last resort of people flying blind. The user’s browser is the last dark room in the architecture, and session replay is a way to put a witness inside it.

> **Field note:** Debugging climbs a ladder. Logs tell you what your servers did; metrics tell you how many and how fast; traces follow a single request across services. All three stop at the browser door. Replay is the rung that begins where the others end — inside the user’s own session. Learn the rungs below it well, and replay will rarely surprise you; skip them, and replay will show you mysteries you cannot explain.

What replay lets you do

Reproduce bugs. Most frontend bug reports arrive as fragments: “the page broke when I clicked the thing.” With replay, the fragment becomes a full sequence — the exact page, the exact click, the exact error, in order. You stop guessing and start watching.

Understand your product. A funnel tells you people abandon at checkout; replay shows you the moment they hesitate, re-read the shipping options, and leave. Watch enough sessions and patterns emerge that no dashboard would show: the button everyone clicks by mistake, the field that confuses everyone, the page so slow that users click elsewhere while it loads.

Support customers faster. This is the quiet superpower. Instead of four emails back and forth — “can you try incognito? can you tell me exactly what you saw?” — an agent opens the

session, sees the problem in under a minute, and replies once with an answer or a fix. The customer feels believed, because they were seen.

Verify your deploys. A release that passes every test can still break real users in ways staging never predicted. Watching fresh sessions after a deploy surfaces anomalies in minutes, not in next week's ticket backlog.

Speak one language. Replay hands developers, support, and product people the same artifact: a user's actual journey. The argument "it works on my machine" dies the moment everyone watches the same machine.

What session replay is not

Replay is not a screen recorder, even though it can look like one. It trades a perfect copy of the pixels for what pixels can never give you: searchability, privacy control at the source, and the ability to treat a session as data.

Replay is not analytics. Analytics counts things; replay shows things. A dashboard can tell you that, for example, 41 percent of visitors leave on the payment page — a valuable fact and a mystery at once; replay shows you what leaving looks like. The dashboard tells you where to look, and replay tells you what you are looking at.

Replay is not a performance monitor. Real User Monitoring (RUM) and similar tools measure page speed, Core Web Vitals, and network behavior across many users, and they are excellent at what they do: telling you how fast, how often, and for whom. But a number is not a story. When a performance metric drops, the question "what did that actually look like for the user?" belongs to replay. Many teams run both: RUM to raise the alarm, replay to read the incident.

Replay is not a substitute for your own logs, metrics, and traces: it watches one side of the system — the browser — and cannot tell you what your backend was doing that moment. When a replay shows something strange, you will often need server-side logs to explain it. Replay plugs the last gap in your observability; it does not replace the rest of it.

One more boundary: replay is not mind-reading. It shows what a user did — not always why. Sometimes the why is obvious; sometimes it is not, and honest teams resist over-interpreting a cursor path. Replay narrows the gap between what happened and why; it does not close it.

The privacy question, up front

No honest chapter about session replay can skip this, early, because replay has a sharp edge. When a site records sessions, it records real people: their hesitations, their typos, the things they almost ordered, the fields they fill in. The very fidelity that makes replay useful — that it

sees what the user sees — is what makes it sensitive. A service that replays everything, forever, to anyone in the company would be a trust and privacy disaster, and in much of the world, including Europe under the GDPR, a legal one.

This concern is not a footnote to the architecture; it is one of the forces that shaped it, and it is why replay services look the way they do. Sensitive fields are masked before data leaves the browser, so a credit card number is never recorded at all. Network payloads are sanitized so tokens and passwords do not ride along. Sessions are encrypted as they travel and as they sit in storage. Retention is bounded: sessions are kept for a defined period and then destroyed, not hoarded. Access is scoped, so an agent sees only the sessions of their own company's users. Visitors are told what is being recorded and asked to consent where the law requires it. None of this is bolted on; each choice reaches deep into how the platform is built.

Carry one rule of thumb from page one: record people the way you would want to be recorded yourself. Capture only what you need, redact what you must, keep it only as long as you promised — and treat every session as a person's afternoon, not a data asset.

A reference implementation — and the road ahead

So far we have talked about session replay in the abstract. The rest of this book makes it concrete by taking one platform apart: **LogNroll**, a real, working, multi-tenant session replay and analytics service in the same family described above. LogNroll is built from the pieces most replay platforms end up with, and this book walks through each in turn: a small recorder that runs inside the customer's website and captures events; a compact encoding for the network; an ingestion gateway that receives events at scale; a message bus that carries them onward; workers that compress and archive finished sessions into object storage; an offline processor that mines sessions for errors, slow requests, and heatmaps; and a player that fetches a session back out of storage and rebuilds it, moment by moment, in your browser. Along the way we look at real code, real payloads, and real operating decisions, honest rough edges included, because the goal is understanding, not marketing.

Threaded through the technical chapters is a story. We follow Candlewood Books, a small fictional bookstore that ships its website in a moment of misplaced confidence and then discovers, through a flood of complaints it cannot reproduce, that it has built a storefront it cannot see. Their struggle is the Tuesday morning of this chapter, lived for real, and their first steps with session replay are our first steps into the architecture. You meet them in the next chapter.

The chapters between travel the life of a session from end to end. Early chapters look at capture: what a session is, how the recorder works, what it records, and how privacy is protected at the source. Middle chapters follow the data out of the browser — how events are encoded, batched, received, queued, archived, and stored — and how archived sessions are

processed into insight and served back out. Later chapters turn to the player that reconstructs time, the product around it, and the business of operating a replay service at scale, including the math of cost and the road ahead. The final chapter returns to Candlewood Books two years on, asking the question most teams eventually ask: given everything you now know, should you build this yourself, or buy it?

By the end, you should be able to look at any session replay product — or at your own web stack — and know what is actually happening underneath: where the data comes from, where it goes, what it costs, and what it can and cannot show you. The best place to begin is where the need begins: not with an architecture diagram, but with a bookstore in the rain, because before you can understand how replay works, it helps to feel why it matters.

Chapter takeaways

- Session replay reconstructs a user's journey from recorded events; it is not screen video, and that difference drives everything else: size, privacy, and searchability.
- A session is one continuous visit by one person on one device, captured as an ordered timeline of events rather than as film.
- Replay exists because modern web applications run most of their logic in the browser — the one part of the system their operators cannot see.
- Replay complements analytics, RUM, and server-side logs: it shows what counts cannot explain, and it works best when the rest of your observability is already solid.
- Replay cannot read minds; it shows what users did, and sometimes still leaves the why to you and your other tools.
- Privacy is not an afterthought — masking, encryption, retention limits, and access control shape the architecture — and the rest of this book walks that architecture end to end through LogNroll and the story of Candlewood Books.

Chapter 2: The Bookstore That Couldn’t See Its Customers

Rain ran down the windows of the little office above the shop in long, patient stripes, blurring the market square into watercolors. Marta Reyes had been at her desk since eight, and the espresso machine had already coughed twice, its way of complaining about Mondays. Below her, Ben Kowalski was dragging cardboard flats across the stockroom, and the smell of paper rose through the heating vents like the shop’s own weather.

She opened the support queue the way a person opens a door they no longer trust. There were 11 new tickets since Friday. She read the first one and felt the familiar small drop in her stomach, a refund form already half-written before she had finished the sentence.

“I placed an order on Thursday and the site said my payment failed, so I tried again. Now I’ve been charged twice and I only have one confirmation email. Can you please help me? I’m not sure what to do.”

She knew what the rest would say without reading further — they all said the same four things now. It charged me twice. Nothing happens when I press Pay. The confirmation email never comes. The site freezes on my phone. In six weeks, the storefront Candlewood had trusted for 10 years had become a machine that swallowed customers in ways no one could see.

The shop on the square

Candlewood Books had begun in 2016 with Maya Okafor, a postal scale, and a card table in the back room of a rented shop that smelled of somebody else’s carpet. Maya had been a children’s-book editor who grew tired of watching good books vanish into the algorithms. Her stubborn theory: a small shop with a sharp eye could sell books anywhere in the country, if only its storefront never closed. The website was the storefront that never closed — open at three in the morning, in kitchens and waiting rooms and on phones held over sleeping babies — and it worked. A decade later, Candlewood sold books, ran a monthly subscription box called the Reading Room, and did something north of €600,000 a year with seven people, a lot of cardboard, and almost no meetings longer than 20 minutes.

The seven people were easy to name. Maya, who still opened every damaged return herself because she liked to know what the post did to books. Priya Nair, the shop’s first real frontend engineer, three patient years untangling a website that predated her. Tom Bakker, part-time backend and DevOps, who kept the servers alive from a farmhouse an hour away. Marta, who ran support with two other people and a spreadsheet she defended like a border.

Ben, who packed the boxes and could wrap a book in paper faster than anyone Maya had hired. And two more people whose jobs shifted with the seasons.

The customers knew the shop the way you know a neighbor: the handwritten note in every Reading Room box, the confirmation email that carried the actual title of the book in its subject line. The old website had been slow — six seconds or more to show a single page, a checkout that felt like filling out forms by candlelight — but it had been *theirs*, and orders arrived, money moved, and nobody cried.

By the autumn of 2025 even Maya had to admit the old site was past saving: patched for nine years by people who had all, at some point, left, absurdly expensive to host, painful on a phone. So, in the gray weeks between one year and the next, they made the decision every small company eventually makes: they would rebuild. Priya would build the new storefront — a modern stack, a fresh checkout, pages that loaded in the blink of an eye — and Tom would move it to better hosting while he was at it. They launched in the first week of March, full of hope and slightly ahead of schedule, which should have been the first warning sign.

For three weeks, the new site was the best thing Candlewood had ever done. Pages appeared instantly. The checkout was clean and fast, with a progress bar that felt like an apology for the old one. Orders crept up. Maya stood in the stockroom doorway one afternoon and said the words she had been saving for months: “I think we’re going to be fine.”

Then April arrived, and the customers started to disappear.

April: the flood

It began with one ticket that could have been anything: a woman in the north of the country, a book for her mother’s birthday, a confirmation email that never arrived — and the birthday was on Thursday. Marta checked: the order was there, the email sent. She apologized, resent it by hand, and closed the ticket with a clear conscience. It was the kind of thing that happened once a month.

The next week there were four like it. The week after that, 11. And around the same time, a second pattern surfaced in the queue, and this one had teeth: *“Nothing happens when I press Pay.”*

Marta read those words and did what she always did: opened the site on her own laptop, added a book to the cart, walked the checkout, pressed Pay. It worked — on her phone, on the shop’s ancient test tablet, in an incognito window, in another browser entirely. So she wrote back with the questions she hated asking: *Which browser were you using? Which device? Could you send a screenshot? Could you try clearing your cache and trying again?* The replies, when they came, were variations on *I did try again. It still didn’t work.* And because Candlewood’s policy was that the customer was right even when the customer was a mystery,

Marta refunded or re-sent or comped the shipping, and closed each ticket feeling that she had paid a ransom and learned nothing.

Priya was having her own version of the same nightmare. She had tested the checkout in every browser she owned and every device she could borrow; her tests all passed. When Marta's tickets arrived with screenshots attached — frozen pages, spinning spinners, error messages in a language that wasn't quite English — Priya opened each one like a detective and found, every time, a page that looked perfectly normal. "It works on my machine," she said one Tuesday afternoon, with the particular weariness of someone who knows how that sentence sounds and hates saying it. Tom, on the other end of the video call, shrugged. "The logs are clean. The API never errors. The payment provider shows nothing unusual. Statistically, some of these people have to be wrong."

Marta looked up from her keyboard. "They're not wrong, Tom. They're just not here."

The tickets climbed through April like a slow tide: more than 40 in the last full week of the month, each one longer than the last, because the easy answers had stopped working and the hard answers didn't exist. Support time per ticket doubled, then doubled again, and the refund column in Marta's spreadsheet grew a second page. The online reviews, four and a half stars for years, began to collect a new species of comment: one star, all caps, always about the checkout. Behind it all, in the analytics Maya checked each morning like a patient with a thermometer, the numbers made no sense. Traffic was up — the relaunch had been good for that — but checkout completion, the number that actually paid for the cardboard, slipped week by week. People were coming. People were leaving. The dashboard could count the leaving. It could not, for anything, show her *why*.

At the Tuesday meeting that week, the four of them did what tired teams do: they looked for someone to blame. Priya blamed the payment provider, whose status page had been a small orange warning for three days. Tom blamed the new frontend, gently, because it was new and therefore suspicious. Priya blamed the hosting migration, which had coincided with the relaunch and never quite been proven innocent. Ben, who had come upstairs with a clipboard and stayed because the meeting smelled like an argument, said the couriers had started asking whether the shop was in trouble. Everyone felt terrible, and the meeting ended early.

Back in her office, Maya looked at the numbers again. Somewhere out there, she thought, a customer is standing at the checkout with a card in their hand, and I cannot see them — not the button they are pressing, not the thing that happens, or fails to happen, when they press it. All I can see is that they were here, and then they weren't.

Mrs. Alvarez

The phone rang at 9:14 on a Friday morning, and Marta answered with her notebook already open.

“Good morning, Candlewood Books.”

“Oh — hello. I’m sorry to call. I hope I’m not interrupting anything important.”

Marta smiled. She knew that opening — every older customer had been taught that phone calls were an imposition. “You’re never interrupting, Mrs. Alvarez. How are you?”

Mrs. Alvarez had been ordering the Reading Room box from Candlewood every month since 2019, without fail. She had told Marta once, in the loose way of someone sharing a small secret, that the box was the one thing she bought for herself — that she looked forward to the last Tuesday of the month the way other people looked forward to birthdays. Marta had never forgotten it.

“I’m fine, dear, I’m fine. It’s the—” There was a pause, the sound of someone choosing words carefully. “It’s the new website. I’m sure it’s me. It’s always me with these things. But I ordered my box, and it said the payment didn’t go through, so I did it again, and now I’ve had two letters from the bank, and I’m worried I’ve paid twice, and I don’t want to be a bother—”

“You’re not a bother,” Marta said. “Let me look.”

She looked. The order was there, and so, unmistakably, was the second one: placed four minutes after the first, same box, same address, both payments taken. She refunded the duplicate while Mrs. Alvarez talked, her voice small and apologetic, explaining that she wasn’t good with the new machines, that the old website had been easier, that she was sorry to make trouble for a little shop that had always been so kind.

“It’s not trouble,” Marta said, and meant it. “It’s done. The extra payment will go back to your bank in a few days, and your box is on its way. I promise.”

“Oh. Oh, that’s—” The relief in the old woman’s voice was physical, like a hand unclenching. “Thank you, dear. You’re so good to me. I’m sorry. I’ll try not to be such a nuisance.”

After she hung up, Marta sat for a long moment. She typed the ticket note the way she had typed 100 others that month: *Duplicate order. Refunded. Root cause: unknown — could not reproduce.* Could not reproduce — as if Mrs. Alvarez’s kitchen, her bank card, her careful double-checking, were a laboratory that had failed to cooperate.

Northside

The email arrived on the second Monday of May, addressed to Maya personally, which was how she knew it was serious before she opened it.

Maya — I need to talk to you about the portal. Can you call me this afternoon? — Dana

Dana Whitfield ran acquisitions for the Northside Public Library system. For three years, Northside had been Candlewood's most important institutional customer: the account that proved a tiny shop could serve a public institution — several hundred titles a quarter for its branches and reading programs. It was steady, large, and, Maya thought privately, the closest thing Candlewood had to a guarantee. When they rebuilt the website, they had folded the library's ordering portal into the new system too — it had seemed so sensible at the time.

Dana answered on the second ring, and Maya heard it in her voice immediately: the careful pleasantness of someone who has decided to be fair and is running out of patience.

"Maya. Thanks for calling. I'll be direct. We're trying to build our summer order — the reading program titles, the branch allocations — and we've been at it for a week. Three of my staff have tried. Every one of them has the same problem: they add titles to the cart, and the cart won't hold them. They add a book, it shows for a moment, and then it's gone. We can't place an order we can't assemble."

Maya wrote *cart empty* in the margin of her notebook. "Dana, I'm so sorry. I didn't know. Have you—"

"We've cleared caches, tried different machines. I've watched one of my staff do it myself — a perfectly normal laptop, a perfectly normal browser." A pause. "Maya, we've been doing business for three years. I want to keep doing business. But the summer order has to go in by the first week of June, or the branches don't get their books before the holidays — and I can't tell my board we lost the summer because a vendor's website couldn't hold a shopping cart. I need to know you can fix this. And I need to know it honestly."

"Give me two weeks," Maya said.

"I can give you 10 days," Dana said. "And Maya — if we have to move the account, I hope you'll understand it isn't personal. It can't be personal."

Maya hung up and sat very still in her chair, counting the days: 10. Below her, the shop door opened and closed, the bell a small, cheerful lie. She thought about the email she would have to write if they lost Northside — polite, grateful, and useless — and about Mrs. Alvarez apologizing on the phone, the refunds, the dashboard that counted the leaving without ever showing the why.

She walked out to the front office, where Priya and Marta were both pretending not to have listened. "That was Northside. The portal's eating their carts. They're giving us 10 days." She looked around the room — at Priya, whose checkout worked on every machine she owned; at Tom on the screen, whose logs were clean; at Marta, who had refunded 40-plus duplicate payments and could not reproduce a single one. "This is the Summer of Blame," Maya said. "Every week, someone blames the browser, or the hosting, or the bank, or the customers, and

nobody — nobody — can actually say what happens when a real person uses our site. We are running a shop where the lights are on and we can't see the customers. I don't know about you, but I am done guessing."

Priya looked up. "Can I show you something?"

The 10-minute integration

What Priya showed them, in the gray light of a rainy Tuesday, was not a fix. It was a way of looking.

"There are tools that record what actually happens in a browser," she said. "Not a video of the screen — something better. They record the events: where the mouse goes, what gets clicked, what gets typed, what the page does. Then you can watch the whole thing back, like a film. If a customer's checkout fails, you can see the failure happen — see what they saw."

Marta leaned forward. "You mean we could have watched Mrs. Alvarez—" She stopped. "We could watch what our customers actually do."

"If we'd had it in April, yes. Probably." Priya turned the laptop around. "There's a service called LogNroll — a session replay platform, made for exactly this. There's a free plan. It takes a few lines of JavaScript to turn on — 10 minutes, maybe less."

Maya was quiet for a moment. "And it records… everything? Card numbers? Passwords?"

"No." Priya shook her head. "That's the first thing you configure. Sensitive fields are masked before they're ever recorded — the card fields, anything that looks like a password. It never leaves the customer's browser in the first place. And visitors get told we're recording, in the footer, with a way to opt out. It's a legal requirement for EU visitors anyway. I wouldn't turn it on any other way."

Maya looked at the rain on the window, at someone below wrestling an umbrella, at the small hopeful face of the laptop. Northside had given them 10 days. Marta had refunded 40 payments that month. And one old woman was still apologizing for being charged twice.

"Do it," she said. "Staging first. Show me."

So Priya did it that same evening, as soon as the shop closed at six. She signed up for LogNroll, created a company called Candlewood Books, and copied the snippet of JavaScript into the storefront's code — a few lines, exactly as promised, with a comment above them so whoever met this code later would know what it was for. On the staging site she walked through a test checkout herself, buying a book she did not need, and there it was in the corner of the page: a small green badge, the recorder's quiet signal that it was alive, watching, working. A dot of green in a browser corner — the smallest thing — but it made her feel, for the first time in weeks, that she was not alone in the dark with the checkout.

She called Maya over, and they watched her test session play back: the cursor moving across the staging page, the little dance of the purchase, the confirmation appearing exactly as it should. Maya watched in silence. Then she said, quietly, “That’s me. That’s my mouse. I can see me.” She laughed, a short surprised sound. “Okay. Turn it on. The storefront tonight, and the portal tomorrow — and send Dana a note that we’re rolling out something to help us see what her staff are seeing, so we can fix the cart for good.”

The recorder went live on the real storefront by eight that night — and it broke nothing, because a good recorder is built to fail silently, a guest in the page that never trips over the furniture. The note to Dana went out at 9:47, and Dana replied at 10:02 with four words: *Good. Keep me posted.*

By the time Marta arrived the next morning, Candlewood had recorded its first real traffic: a few hundred anonymous sessions — people browsing the bestseller list at two in the morning, reading the Reading Room page, adding books to carts and, mostly, buying them. Mostly.

The replay

They gathered around Priya’s desk at 9:40 that morning, the four of them plus Ben, who had come upstairs to see the fuss and stayed. The rain had stopped for the first time in a week, and the light through the windows was the particular yellow of a town drying out. The LogNroll session list showed the night’s traffic as a column of small rectangles — every visit, every device, every page.

Priya had filtered for one thing: sessions that reached the checkout and did not complete. There were more of them than she wanted to see. She clicked one — a session that had started at 8:07 the previous evening, on a phone, and had ended 22 minutes later.

The player opened. And then the room went quiet, because there it was: the Candlewood checkout, rendered on the screen exactly as a customer had seen it the night before. A narrow phone-shaped page. The order summary — three books and the Reading Room box. A shipping address in a town Marta had never heard of, typed carefully, one field at a time.

The customer’s thumb moved slowly. They scrolled up to re-read the order, scrolled down, hesitated over the delivery options — chose the slower one, the cheaper one, the small economy that said something about who they were. They reached the payment section. The cursor — the ghost of their thumb — hovered over the Pay button.

And pressed it.

Nothing happened.

On the screen, the button flickered — a gray flash so quick it was almost nothing — and then sat there, unchanged. The customer waited. The little circle that should have meant *working*

did not appear. They pressed again. Nothing. They scrolled up to check the order summary, as if the problem might be there, scrolled back down. Pressed a third time. The button flickered again, politely, and did nothing at all.

For a long moment nobody in the room breathed. Marta watched the stranger's thumb hover over the dead button, watched them wait, watched them scroll away and come back, the way you return to a door that won't open. At 8:29 the customer gave up. The page went dark. The session ended.

"That," Marta said, very quietly, "is ticket 1,283. 'Nothing happens when I press Pay.' That's what it looks like."

"It's a person," Maya said. "A real person, in a real kitchen somewhere, at eight o'clock at night, trying to give us —" she leaned closer, counted the order summary — "42 euros. And we made it impossible. We built a door that doesn't open and then blamed them for standing in front of it."

No one said anything. On the video call, Tom had gone very still.

"It's like standing behind their chair," Marta said. "I've asked 100 customers to describe this to me, and none of them could, and now I've seen it. We still don't know *why* the button didn't work — but for the first time in two months, we know it wasn't them."

Priya clicked the session list again. "Let's look at another one."

They watched three more failed checkouts that morning — different phones, different hours, different books — and each ended the same way: at the Pay button, in silence, with a customer who eventually went away. They also watched one that succeeded: a woman in the next town over who breezed through the same checkout in 90 seconds. Same site. Same button. Same night. One customer sailed through while the others stood at a door that wouldn't open — and nobody had to ask them to describe it.

Maya said little for the rest of the morning. Back in her office, with the window open, she thought about the strange intimacy of what she had just seen: a stranger's evening reconstructed in a browser tab — the hesitation at the shipping options, the careful second try at a button that had already failed her. And the machinery that had brought that evening to her screen — the recorder riding along in the page without breaking it, the small events carried through the night and rebuilt into a story — felt like a witness protection program in reverse: an invisible chain of custody that had carried one person's ordinary frustration across the internet and laid it, gently, on her desk.

She picked up her notebook and wrote three words under the date: *We can see now.*

Then she wrote four more: *Now we find the why.*

What happens next

What replay showed Candlewood that Wednesday morning was the beginning of the answer, not the answer itself. They could see the failure now — the click that did nothing, the customer who walked away — but seeing is not understanding, and the button that refused to open still had its reasons. Finding them would mean taking the replay apart: the events that recorded the click, the error that fired in the console a heartbeat later, the network requests that went out and the ones that never did, the exact shape of the page the customer's browser had built. Every layer of that machinery is a chapter in this book.

By summer, Candlewood would find its bugs — the checkout that failed in one browser and not another, the double charges that came from a button pressed twice, the library portal whose carts would not hold, the slow pages that froze on phones. They would fix them, keep their oldest client, and redesign their homepage before the holidays. But that is the second half of the story, and this book has a long way to travel before it returns to the shop on the square.

What matters is what happened on that rainy Wednesday morning in May: a small bookstore learned to see its customers. A stranger in a kitchen, pressing a button that would not open, had been seen — really seen, as if someone had been standing behind her chair. And the machine that made it possible, the quiet chain of record and transport and storage and reconstruction that carried her session from her phone to Priya's screen, is the machine this book is about.

The button had done nothing. Why it had done nothing was about to take them — and this book — through every layer of the system between a customer's tap and a team's understanding. The first replay had opened the door. Now they had to learn what was behind it.

Chapter 3: Anatomy of a Session

What a session is, what it contains, and where it lives as it travels through the LogNroll platform.

When the Candlewood Books team first opens the LogNroll dashboard and sees a list of sessions, each row is a promise: pick one, and you can watch a real shopper move through the real storefront. Before the team can trust that promise, they need to know what a row actually is. The word "session" gets thrown around casually, so this chapter pins it down. A session is the core unit of session replay — the thing that is recorded, transported, stored, analyzed, and eventually played back. Understand the anatomy of a session and you understand the shape of the entire pipeline that the rest of this book walks through, service by service.

One Visit, One Record: What a Session Is

A **session** is one continuous period of activity by one user on one device. It begins when recording starts and ends when the user goes away: they close the tab, navigate elsewhere, or simply stop doing anything for long enough that the platform gives up waiting and closes the record for them. One session never mixes two users or two devices, because a session is born inside a single browser tab and lives and dies with the events that tab produces.

That "single tab" detail matters more than it sounds. A person can have the storefront open in three tabs while also checking the library portal on their phone. Each tab is a separate recording context, so each becomes its own session. Refresh the page and the session continues, because the tab is still alive and its identity is preserved. Close the tab and the session ends; open the site again and a brand-new session begins. The same human can therefore produce several sessions in one afternoon, and the platform treats them as separate records, linkable later through a shared device identifier and, when known, through their user identity.

A session is also bounded by patience, not just by the visitor's behavior. Users leave tabs open overnight, walk away mid-checkout, or get interrupted for an hour. The platform cannot wait forever for the next event, so it applies timeouts: a session that has been quiet for long enough is closed and moved along the lifecycle we describe in a moment. From the platform's point of view, a session is not "while the human was present"; it is "while the recorder was actively sending us their events." That definition is precise, machine-checkable, and good enough for the product's purposes — which is exactly the kind of definition an engineer should want.

The Session Id: `NEW`, Then Assigned

Every record needs an identity, and sessions are no exception. The interesting wrinkle is that a session id cannot exist until the platform has seen the session, yet the recorder must start sending before it knows whether the platform has ever seen this tab before. LogNroll resolves this with a two-step contract that looks almost too simple to be real: the recorder starts with the sentinel value `sid = 'NEW'`, and the receiver answers with the real id.

Here is the mechanism, which we verified in the recorder source. The recorder keeps its session id in **sessionStorage** under the key `lognroll`. sessionStorage is scoped to the tab and survives page reloads, but it is cleared when the tab closes. When the recorder boots and finds no stored id, it asks for one by pretending it has the id `NEW`:

```
// From the recorder: the session id lives in sessionStorage for the tab's life.
// When the tab has no id yet, the recorder starts with the sentinel 'NEW'.
getSid() {
  return sessionStorage.getItem('lognroll') || 'NEW';
}

setSid(sid: string) {
  sessionStorage.setItem('lognroll', sid);
}
```

The recorder's first contact with the platform is a POST to `receiver.lognroll.com/lognroll/{companyId}/NEW`. The receiver sees `NEW`, creates a brand-new session record, and returns the real session id — the hexadecimal identifier of the new Mongo document — as the body of the response. The recorder adopts that id, stores it back into sessionStorage, and from then on sends every batch to `/lognroll/{companyId}/{realId}`.

Two details make this contract robust rather than cute. First, the handshake is best-effort: if the very first request fails because the network is flaky, the recorder does not stop; it simply keeps posting later batches to `.../NEW`, and whichever batch arrives first creates the session and returns the id. The session self-heals. Second, the receiver checks whether a posted id still refers to a live session. If the recorder comes back with an id whose session has already been finished server-side — say the user returned after a long idle period — the receiver quietly starts a fresh session and hands back its new id, which the recorder adopts in turn. The sentinel `NEW` is therefore not a hack; it is a small, honest protocol for "I do not know my name yet, please give me one," and it survives network failure and server-side timeout alike.

The Lifecycle: `ACTIVE`, `IDLE`, `FINISHED`, `REMOVED`

A session record moves through four official states, and the exact strings matter because every service in the platform reads and writes them in a shared Mongo collection called `sessions`:

```
ACTIVE → IDLE → FINISHED → REMOVED
```

`ACTIVE` is the state of a session that is receiving events right now. When the receiver creates a session — the `NEW` handshake above — it writes the status `ACTIVE`. Every subsequent batch of events refreshes the record: the receiver updates the session's last-seen timestamp, and if the batch contains user interaction, it stamps the session `ACTIVE` again and notes the moment of that interaction. As long as a shopper keeps clicking, scrolling, and typing, their session stays `ACTIVE`.

`IDLE` is the intermediate state for a session that has gone quiet but might still come back. When the receiver processes a batch that contains no user interaction — traffic without clicks, scrolls, or keystrokes — and the session's last real interaction is still recent, the receiver may demote the session to `IDLE`. The next interaction promotes it straight back to `ACTIVE`. Think of `IDLE` as the platform saying "we have not seen the user do anything in a moment, but we are not giving up on them yet."

`FINISHED` is the state that matters most to the rest of the pipeline, because it is the trigger for archival. A dedicated job — LogNroll's session-status-job, a cron that runs every minute — walks the `ACTIVE` and `IDLE` sessions and finishes the ones that have been quiet too long. The thresholds are real, read from the job's Go source, and they are short enough to feel surprising:

- Thirty minutes since the session's last recorded interaction, or
- Two minutes since the session record was last updated at all, or
- Three hours since the session started, as a hard ceiling.

When any of those conditions holds, the job sets the status to `FINISHED`, records an end time, and computes a duration. The exact borderline behavior — which condition wins when two apply at once, and why the two-minute rule exists alongside the thirty-minute one — is a story we will tell properly in the storage chapter. For the anatomy, what matters is the outcome: a session does not linger forever. Within a few minutes of the user going silent, the session is `FINISHED` and ready for its next phase.

`REMOVED` is the final state, and it means the data is gone. Sessions do not reach it quickly. A second job — the remover — claims `FINISHED` sessions older than thirty days, deletes their archived data, and marks them `REMOVED`. It runs on a short schedule in production (every few minutes) so deletions stay timely; development environments run it hourly. There is also a failure path: if something goes wrong during archival, a session can be flagged `FAILED` before it is ultimately removed, so operators can see at a glance that a session never made it into the archive. The removal cadence — thirty days — is a product decision about retention and cost, and we will return to it.

Notice who is *not* in this story: no human closes a session, and no single service owns the whole lifecycle. The receiver activates and idles; the status job finishes; the worker archives; the remover deletes. Each transition is a small, idempotent database write guarded by locks, and each service only touches the states it understands. That division of labor is the architecture in miniature, and it is why the platform can keep millions of sessions moving without a central orchestrator.

What a Session Contains

Strip away the dashboard, and a session is two things: an **ordered timeline of events**, and a **record of metadata** that describes who produced those events, from where, and on what device.

The timeline is the soul of the replay. Every captured interaction becomes a **log point**, a single structured event, and log points carry six fields that we will dissect in the protobuf chapter: a timestamp, a type, a payload, a processing version, and two sequence numbers — one ordering events inside the session, one ordering them inside their batch. The payload for most types is JSON, which keeps the schema flexible; only a handful of types carry raw data.

The type field selects from exactly fifteen event kinds, defined once in the shared protobuf contract (`LogPoint.proto`) and copied, byte for byte, into every service that touches log points. They are the vocabulary of everything that happens on a page:

Value	LogType	Kind of event
0	NAVIGATION	URL or route change: full page loads and single-page-app navigation
1	MUTATION	DOM changes: nodes added or removed, attributes and text modified
2	CLICK	Pointer clicks with coordinates and the target element’s identity
3	INPUT	Value changes in input fields
4	MOUSE_MOVE	Pointer movement, throttled and batched to keep volume sane
5	LOG	Console output: log, warn, error, and friends

Value	LogType	Kind of event
6	NETWORK	Fetch and XMLHttpRequest calls with method, URL, status, and timing
7	SCROLL	Scroll positions, absolute and relative to the element that scrolled
8	KEYBOARD	Key events, batched, with modifier keys ignored
9	FORM	Form field data, debounced, with sensitive fields excluded
10	META	Page metadata: URL, viewport, browser, and capability flags
11	IDENTIFY	The user's identity, attached when the site calls identify
12	STYLES	Style sheets and dynamic styles needed to render the page faithfully
13	PING	Heartbeats used to tell a live session from a dead one
14	PERFORMANCE	Memory statistics and long-task timings

Fifteen types is a small enough set to memorize and a rich enough one to rebuild a page visit. Click, scroll, keyboard, and input events describe what the user did; mutation and style events describe what the page did in response; navigation and meta events describe where the user was; log, network, and performance events describe what the browser was doing underneath. Put them all in timestamp order and you have a complete, searchable account of a visit — which is the whole point of replay, and the reason the pipeline exists.

Around the timeline sits the metadata record. The `sessions` document in Mongo is not a list of events; it is a summary that the dashboard can render instantly without touching the archive. It remembers which company the session belongs to, the device id that produced it, the browser and operating system parsed from the user agent, the IP address and the country and city derived from it, the pages visited, the start and end times, the duration, the current status, and — once the site has identified the visitor — the user's id, name, and email. Errors and network calls are not stored inline; they are derived later, by the processor, into their

own collections, and joined back to the session when you open it. So a session row in the dashboard is a curated summary, and the full timeline is the raw material behind it.

The Pipeline, End to End

A session does not simply “exist” in the database. It is born in a browser, carried across the internet, parked on a message bus, filed into an archive, analyzed, and finally served back to a player. The following diagram is the canonical data flow of the LogNroll platform, and nearly every chapter in this book is a zoom into one of its boxes:

```
Browser (customer website)
   │  embeds SDK (@lognroll/lib) → loads recorder bundle from logger.lognroll.com
   ▼
[LOGGER]  captures mouse/scroll/keyboard/input/form/DOM/console/network events
   │  → protobuf LogPoint[]  (LogPoint.proto, 15 event types)
   ▼
[RECEIVER]  POST /lognroll/{companyId}/{sid}   (receiver.lognroll.com)
   │  creates/tracks session in Mongo; encrypts points
   ▼
[NATS JetStream]  subject sessions3-{env}.{sessionId}   (stream sessions3-*-stream)
   ▼
[WORKER]  pull-consumes JetStream → decrypt → unzip → re-chunk → zip
   │  → archives to S3/Spaces (sessions/{sessionId}/…)
   ▼
[SESSION-PROCESSOR]  locks session, reads S3 archive, runs analytics
   │  → sessionErrors / backendRequests / heatMapClicks / scrollHeat (Mongo)
   ▼
[PLAYER-API]  GET /api/get/{id} → decrypts S3 archive → binary protobuf
   ▼
[PLAYER]  (Angular) decodes LogPoints → DOM/console/network replay + heatmaps
```

Read the diagram top to bottom and you can see the session’s whole biography in miniature. The recorder on the customer’s website — the subject of the next chapter — captures events and encodes them as protobuf log points. The receiver, an ingestion gateway, accepts those points, creates or updates the session in Mongo, encrypts the payloads, and publishes them to a NATS JetStream subject. The worker pulls the session’s messages off the bus, decrypts and uncompresses them, re-chunks the event stream, and files the result as zipped archives in S3-compatible object storage. The session processor then reads the archive and derives the analytics that power error digests, backend-request analysis, click heatmaps, and scroll heat. When someone opens the session in the dashboard, the player API fetches the archive, decrypts it, and serves binary protobuf back to the Angular player, which decodes the log points and reconstructs the visit frame by frame.

Each hop exists for a reason that maps to a chapter. The recorder is where capture and privacy decisions happen. The wire format matters because it determines how much bandwidth a million sessions cost. The receiver is the first touch of every event and the place where hot-path discipline is enforced. The message bus decouples the bursty ingestion from the slower archival work. The worker and the object store are where sessions wait, cheaply,

until someone needs them. The processor is where data becomes insight. The player API and the player are where insight becomes something a human can watch. And the lifecycle states we described above are the thread that ties the hops together, because every service reads and writes the session's status to decide whether its job is done.

A Session Is Data, Not Video

It is worth stopping on a phrase that will recur throughout this book: a **session is data, not video**. The replay player makes a session look like a screen recording, but nothing on the LogNroll platform ever records pixels. The recorder captures events — coordinates, keystrokes, DOM changes, console messages, network calls — and the player later *reconstructs* the page from those events, redrawing the DOM and moving a virtual cursor along the recorded path. The illusion of video is the product of careful rendering, not of a camera.

Why go to that trouble? Three reasons, and they are the same three reasons the entire industry converges on this design. The first is size: a stream of coordinate and mutation events is orders of magnitude smaller than a video of the same interaction, which matters when a platform stores millions of sessions and replays them over ordinary internet connections. The second is fidelity to meaning: a video shows a spinner; an event stream shows the exact network request that returned a slow response, the exact console error that accompanied it, and the exact element the user was trying to click. The third is searchability and structure: events can be filtered, aggregated, and analyzed — which is what makes heatmaps, error digests, and "show me every session where checkout failed" possible at all. You cannot query a video. You can query data.

Data, of course, has its own consequences, and honesty requires naming them. Reconstruction is not perfect: exotic canvases, video elements, cross-origin iframes, and CSS animations replay approximately, not exactly, and the player chapter will spend real time on those limits. And because sessions are data about real people, they inherit privacy obligations that a discarded video file would not obviously carry — which is why masking, consent, and retention are first-class architectural concerns rather than afterthoughts. The trade is deliberate: the platform gives up pixel-perfect fidelity to gain scale, meaning, and accountability, and that trade is exactly what makes replay a debugging and product tool instead of a surveillance toy.

> **Story checkpoint — Candlewood Books:** A few days after the first replay made its way around the office, Marta Reyes, the support lead, sits down with Priya Nair to look at a handful of sessions from a returning customer who had accepted the site's consent notice. Marta had expected shaky screen recordings; what she saw instead was a timeline she could read like a report — a hover, a pause, a click on the shipping estimator, a longer pause, a scroll back up, silence. "So this is the whole visit," she said. "Everything they did, in order, until they left." Priya nodded and pointed at the dashboard row above the player: the session had started as ACTIVE and had been marked FINISHED a few minutes after that last scroll. "That's the shape of it," Priya said. "One visitor, one tab, one record — from first event to silence." For Marta, it was the moment the abstraction clicked: a replay is not a movie of a customer; it is the customer's journey, stored as data, waiting to be read.

Chapter takeaways

- A **session** is one continuous period of activity by one user on one device, born in a single tab, bounded by inactivity timeouts rather than by the visitor's intentions.
- The session id follows a two-step contract: the recorder starts with `sid = 'NEW'`, and the receiver creates the session and returns the real id, which the recorder keeps in sessionStorage under the key `lognroll`.
- Sessions move through four official states — `ACTIVE`, `IDLE`, `FINISHED`, `REMOVED` — maintained by different services: the receiver activates and idles, a minute-cron finishes, the worker archives, and a removal job deletes after thirty days.
- A session contains an ordered timeline of fifteen log point types plus a metadata record describing the visitor, their device, their location, and the pages they saw.
- The platform pipeline moves that data through recorder, receiver, message bus, worker, archive, processor, player API, and player; each hop is a chapter of this book.
- A session is data, not video: reconstruction beats recording on size, fidelity to meaning, searchability, and privacy, at the cost of approximate rendering for exotic page content.

Chapter 4: The Recorder SDK: From Snippet to Session

How a website turns on recording with a few lines of JavaScript — and why those lines are safe to ship.

Every session in the LogNroll platform starts the same way: on somebody's website, a small piece of JavaScript decides to start recording and asks the platform for a session id. Everything downstream — the receiver, the message bus, the archive, the processor, the player — only ever works with what that script sends. The recorder is therefore the most important and the most constrained component in the whole system, because it runs inside your customers' browsers, on their machines, under their network conditions, and it must never, ever break the page it is observing.

This chapter follows Priya Nair through the first half hour of her LogNroll integration: getting a company key, adding the SDK, loading the recorder, and watching a session appear. By the end you will know exactly what "a few lines of JS" actually do, what runs where, how a session gets its id, and — just as important — how the design makes it safe to put recording code on a storefront where a single thrown exception could cost real money.

The Two Ways In: Package or Snippet

LogNroll's recorder is distributed as two layers that work together. The first layer is the **SDK**, published on npm as `@lognroll/lib` — a thin wrapper that site owners import and configure. The second layer is the **recorder bundle**, the larger capture engine that does the actual work, which the SDK loads from the LogNroll host `logger.lognroll.com`. Site owners never touch the recorder bundle directly; they interact with the SDK, and the SDK handles the rest.

For an application built with a bundler, adding the SDK is an npm install plus an import. The package's default export is a ready-made instance, so there is no `new` call and no manual wiring:

```
import lognroll from '@lognroll/lib';

// Turn on recording for this page. The key identifies your project and
// appears in every upload URL as /lognroll/{companyId}/{sid}.
lognroll.initSession('candlewood-storefront');
```

That is the whole integration for a modern JavaScript app: one import, one call. But LogNroll serves customers who run anything from a React storefront to a WordPress site to a static brochure page, so the SDK is also built as a UMD bundle that attaches a `LognRoll` global

when loaded with a plain script tag. A site without any build step can include the SDK file directly and call the same method on the global:

```
// No bundler required: the SDK ships a UMD build exposing a LognRoll global.
// Load the file wherever you host your copy, then initialize.
LognRoll.initSession('candlewood-storefront');
LognRoll.identifyUser('usr_8f2c', { name: 'Dana Whitfield', email: 'dana@example.com' });
```

Whichever route a site takes, the code that ends up running is the same. The SDK records your company key, and that single string is the thread that ties everything you record to your account: the recorder sends it with every batch, the receiver uses it to decide which company the session belongs to, and the player API later uses it to enforce that only your team can watch your sessions.

The Company Key, By Any Name

Look closely at the SDK's own type declarations and you will find a small naming inconsistency worth knowing about, because it explains how the pieces fit together. In the published type definitions, the first argument of `initSession` is declared as `apiKey`. In the implementation, the same argument is stored as `companyId` — and `companyId` is the name that travels through the rest of the platform. The receiver's route is `POST /lognroll/{companyId}/{sid}`; the session document in Mongo carries a `companyId` field; and the dashboards you use to manage access are organized around companies and teams. The SDK calls it a key because that is how the docs describe it to newcomers; the platform calls it a company id because that is what it is. They are the same value, and when you read LogNroll code or this book, translate freely.

Where does the key come from? It is issued for your project when you sign up and create a company in the LogNroll dashboard. For Candlewood, that means Priya creates one company and intends to use the same key across the storefront and the library portal, so all their sessions land in one place where the team can filter by URL. In the example snippets above, `'candlewood-storefront'` stands in for the real opaque key; treat it as a stand-in for the string you would copy out of your own dashboard.

What init Actually Does: One Call, Three Jobs

Behind its innocent signature, `initSession` performs three distinct jobs, and each one is a deliberate design decision. The implementation lives in the SDK's core module, and it is short enough to read in full.

First, the SDK publishes itself on the page: it sets `window['lnr']` to the SDK instance. `lnr` is the agreed meeting point between the SDK and the recorder bundle. The recorder bundle, when it later runs, reads its configuration from that global — the company id, the device id,

the options object. Nothing is passed by function call between the two layers, because they are loaded as two separate scripts that may not even be in the same bundle graph; a well-known global is their interface.

Second, the SDK establishes a stable device identity. It looks for a device id in localStorage under the key `lognroll_device_id`; if none exists, it generates a random UUID and stores it there. This id is what lets the platform later recognize that two different sessions came from the same browser, even before any user is identified. The recorder will send it as an `X-LogNroll-Device-Id` header on every batch.

Third — and this is the step that makes everything real — the SDK loads the recorder bundle. It creates a script element and points it at the recorder file:

```
// From the SDK core: append the recorder script to the document head.
// ?cc= is a cache-buster so a fresh bundle is fetched after each deploy.
const LOG_FILE = 'https://logger.lognroll.com/logger.lnr.1.0.1.js';
const script = document.createElement('script');
const now = Date.now();
script.src = (config.logFile || LOG_FILE) + '?cc=' + now;
script.async = false;
document.head.appendChild(script);
```

Three details here repay attention. The default recorder URL is the real constant from the SDK source: `https://logger.lognroll.com/logger.lnr.1.0.1.js`. The `?cc=` query parameter is a cache-buster built from the current time, so that after LogNroll ships a new recorder version, browsers do not keep executing a stale cached copy for weeks — a real operational concern when a bug fix in the recorder must reach every customer's site. And the configuration supports a `logFile` override, which is how a customer can self-host the recorder bundle or point a staging environment at a test build. The same override pattern applies to the receiver base URL, which the recorder reads from the options when deciding where to send batches.

The script is appended to the document head, meaning the browser fetches it in the background while the rest of the page carries on. The recorder does not need to be present for the page to render, and nothing in the host application ever depends on its return value. That is the load-order contract in one sentence: the recorder arrives late, runs independently, and its absence changes nothing about the page's behavior.

The Recorder Bundle: What Actually Runs

The file loaded from `logger.lognroll.com` is the **recorder bundle**: the compiled capture engine, a single JavaScript file containing the logger's whole working set — event listeners, DOM observation, console and network capture, session management, and the code that talks to the platform. Versioned file names like `logger.lnr.1.0.1.js` encode the bundle's release line; a parallel build, `full.lnr.1.0.1.js`, packages the same recorder for contexts

where the full engine is injected whole, which is exactly what the Chrome extension does (we will meet it shortly).

When the bundle executes, it boots a logger that wires up the capture subsystems — the DOM observer, the mouse, keyboard, form, and scroll trackers, the console and network loggers, the performance observer — and registers listeners for page navigation. From the very first moment it is running, events begin flowing into an in-memory queue. But here is the architectural subtlety: the capture code runs on the page's main thread, where it must be as cheap as possible, while the expensive work — serializing events into binary protobuf, batching, and POSTing them across the network — is deliberately moved off the main thread. That division is the subject of the next section, and it is the single most important performance decision in the entire client.

The Web Worker: Keeping the Checkout Fast

A session replay recorder that made the host page janky would be worse than useless, and the heaviest operations in recording are not the listeners; they are the plumbing: turning thousands of small events into compact binary messages and pushing them over the network while the user is trying to pay for something. LogNroll moves that plumbing into a **Web Worker**, a second script context that the browser runs in parallel with the page, on a separate thread, without access to the DOM but also without the ability to block rendering.

The recorder creates its worker from an in-memory blob of worker code rather than from a separate network request — a trick that keeps everything in one loaded bundle:

```
// From the recorder: spin up the batching worker from an in-memory blob.
const workerBlob = new Blob([workerCode], { type: 'application/javascript' });
const workerUrl = URL.createObjectURL(workerBlob);
this.worker = new Worker(workerUrl);
```

The main thread sends the worker a single `init` message carrying everything the worker needs: the session id (possibly still `NEW`), the device id, the company key, the receiver base URL, and the sanitized options. From that point on the division of labor is strict. The main thread collects events, batches them on a timer — the recorder's constants declare a batch size of 200 events and a flush delay of 100 milliseconds — and hands full batches to the worker. The worker owns the outbound queue: it encodes each batch into protobuf, chunks the stream when a batch would exceed roughly 1 MB, and POSTs chunks to `receiver.lognroll.com/lognroll/{companyId}/{sid}` with a `Content-Type` of `application/x-protobuf` and the `X-LogNroll-Device-Id` header.

Because the worker is a separate thread, network serialization and I/O never contend with the page's rendering, scrolling, or click handling. And because the worker, not the page, owns the queue, a slow network cannot make the page wait: batches pile up in the worker's

memory and drain as the connection allows. The queue has a bounded retry policy — a handful of attempts with exponential backoff starting around a second and capping at thirty seconds — after which a stubborn chunk is dropped with a console warning rather than retried forever. Data loss is possible in the worst case, and the design chooses to make that loss loud instead of silent, but it never makes the page slow. The golden rule, which we will return to throughout the book, is that the recorder must be invisible: the checkout must feel exactly as fast with recording on as with recording off.

Starting a Session: `NEW` and the sessionStorage Key

With the worker running, the recorder needs a session id before it can send anything meaningful, and this is where the `NEW` handshake from the previous chapter plays out on the client side. The recorder consults **sessionStorage** under the key `lognroll`. sessionStorage is scoped to the tab and survives reloads, so it is the perfect place to remember "which session does this tab belong to?" If a stored id exists, the recorder resumes that session. If not, the recorder's `getSid` returns the sentinel `'NEW'`, and the worker's very first act is to POST an empty body to `/lognroll/{companyId}/NEW`.

The platform answers with the real session id in the body of the response. The worker adopts it — accepting the response body only if it looks like a genuine id — and posts a `session-id` message back to the main thread, which stores the id into sessionStorage under `lognroll` and remembers it for the life of the tab. From then on every batch goes to the real id. Reload the page and the tab still holds the id, so the same session record continues; close the tab and the id vanishes with the sessionStorage, so the next visit starts a fresh `NEW` conversation. If the server has already finished the old session during a long absence, the receiver detects it and hands back a brand-new session id, which the recorder adopts just as easily. The client never has to know which of those cases it is in; its only job is to ask with `NEW` and trust the answer.

identify(user): Attaching a Name to the Session

An anonymous session tells you what happened on the page but not who it happened to. To close that gap the SDK exposes `identifyUser`, which takes a user id and an optional set of traits — in the shipped type declarations, a name and an email:

```
lognroll.identifyUser('usr_8f2c', {
  name: 'Dana Whitfield',
  email: 'dana@example.com',
});
```

The mechanics are deliberately asynchronous and tolerant of failure. `identifyUser` does not send anything itself; it just records the user id and traits on the shared `lnr` global. The recorder, on its regular flush cycle, notices that a user id has appeared and emits a single

`IDENTIFY` log point carrying the id, name, and email. The receiver, in turn, applies that identity to the session document — but only once. A session that identifies its user early will not have its identity overwritten by a later, conflicting `identify` call, because the receiver records the fact that an `IDENTIFY` event was already seen. The result is a session that can be found in the dashboard by user name or email, and a device whose future sessions can be linked back to the same person. Candlewood calls `identifyUser` on login and at checkout for known customers; everyone else stays anonymous, which is exactly the privacy posture the platform is designed around, and which later chapters will examine in detail.

The Chrome Extension: Recording on Demand

Not every recording happens on a site that has integrated the SDK. LogNroll ships a Chrome extension — a Manifest V3 extension whose popup asks for a company key and then injects the recorder into whatever tab you are looking at. It is a manual injector: rather than the site loading the recorder, the extension pushes a prebuilt recorder bundle (`full.lnr.1.0.1.js`) into the active tab on demand, using Chrome's `scripting` API, and the recorder boots against the company key you typed.

Why does this exist? Two reasons, both practical. First, it is the fastest possible way to try the product: install the extension, open any page of your own site, type your key, and watch a session appear in the dashboard — no code changes, no deploy, no waiting for a release. Second, it is a developer and QA tool: Priya uses it to record her local development server and the staging environment before the snippet is anywhere near production, because injecting into `localhost` requires no CSP changes and no consent plumbing. The extension README is explicit about the workflow: navigate to a page, click the extension icon, enter the company id, and inject.

The same property that makes the extension convenient makes it worth a caution. It can record any tab you point it at, and its manifest asks for broad host access. That is appropriate when the tabs are your own staging and development pages; it would be deeply inappropriate pointed at sites you do not operate, and no legitimate use of the platform involves recording other people's pages without their knowledge. The right mental model is "a manual injector for development and testing," not "a way to record the internet."

Failure Tolerance: Recording Must Never Break the Host

The whole client design so far — late script loading, an off-thread worker, best-effort handshakes — is in service of one non-negotiable rule: **recording must never break the host site**. A recorder that occasionally loses data is an annoyance; a recorder that throws an exception into the page's event handlers, blocks rendering, or slows the checkout is a liability

that no storefront can accept. The LogNroll client treats that rule as a hard constraint and defends it at every layer.

Start with loading. The recorder is fetched in the background and executed independently; if the fetch fails, the page simply continues without it, and no host code ever depended on its presence. Storage access is wrapped in try-catch: in a private window or a locked-down browser where localStorage is unavailable, the device id falls back to an empty string rather than throwing, and the SDK keeps working without a stable device identity. Web Worker support is checked before the worker is created; if the environment lacks workers, the recorder logs a warning and continues capturing rather than failing outright. Console capture, which wraps the browser's own `console` methods, is the riskiest interception of all — a broken wrapper would corrupt every log statement on the page — so the wrapper calls the original method first, inside its own try-catch, and any failure in the capture path degrades to the original behavior. Network capture is structured the same way: sanitizers that throw are caught, and requests the SDK cannot process safely are passed through untouched.

The same discipline governs the outbound path. Sending failures never propagate into the page; they are handled inside the worker, which retries with backoff and reports progress through console warnings prefixed with `[LognRoll]`. If the receiver is down for a few seconds, the user of the storefront will never know. Even the worst case — a batch dropped after exhausting its retries — surfaces as a warning for the developer and a gap in the recording, never as an error on the page.

What Can Still Go Wrong

Honesty compels a list of the ways recording can fail anyway, because every one of them is a configuration problem on the host site rather than a bug in the recorder. The most common is a **Content Security Policy**. A strict CSP that lists permitted script sources will block the recorder script unless `logger.lognroll.com` is added to `script-src`, and will block the batches unless `receiver.lognroll.com` is added to `connect-src`. Because the worker is created from a blob URL, a policy that restricts `worker-src` can prevent the worker from starting even when the main bundle loads. The fixes are mundane: add the two LogNroll hosts to the policy, allow the blob scheme for workers, and test in a staging environment that mirrors production's headers.

Next come **ad and tracker blockers**. Recorder traffic to a third-party analytics host looks, to a generic blocker, exactly like the tracking scripts many sites deploy — and some lists will block or throttle it. There is no code fix for that on LogNroll's side; the recorder simply does not run for those visitors, and the platform treats the missing data as an accepted cost of respecting the visitor's choice. Third, **network conditions**: corporate proxies, captive portals, or a receiver that is briefly unreachable all manifest as the retry-and-warn behavior

described above. Fourth, and worth stating plainly: **the handshake must be allowed**. If the very first `POST .../NEW` is blocked, no session id ever arrives and nothing is recorded; this is the most common "why is nothing showing up?" cause, and the first thing to check in the browser's network tab is whether that request completed.

Debugging a silent recorder is therefore a short checklist: is the script request to `logger.lognroll.com` present and successful? Is the `NEW` handshake reaching the receiver? Are there `[LognRoll]` warnings in the console? Are the CSP and blocker extensions letting the traffic through? Nine times out of ten, the answer lives in one of those four places — and the tenth time, the console warnings will say exactly what was dropped and why.

> **Story checkpoint — Candlewood Books:** Priya created the company in the LogNroll dashboard, copied the key, and added the SDK to the storefront's staging build first — the checkout flow, where the complaints were loudest. On staging she walked a test checkout herself, watched her own session appear in the dashboard within seconds, then called Maya Okafor over and played it back. Maya asked the question Priya had been waiting for: "So if this breaks, could it break the shop?" Priya answered by demonstrating instead of asserting — she turned off the network and reloaded the staging page; the storefront rendered instantly, with a polite warning in the console and no session recorded. Maya nodded and gave the order: the storefront tonight, the library portal tomorrow. The recorder went live on the real storefront the same evening, and by the next morning the first real sessions of Candlewood's actual customers were flowing into the dashboard, waiting to be watched.

Chapter takeaways

- The recorder ships as two layers: the thin `@lognroll/lib` SDK that site owners configure, and the recorder bundle it loads from `logger.lognroll.com`.
- `initSession` does three things: publishes the SDK on `window['lnr']`, establishes a stable device id in localStorage, and appends the recorder script with a cache-buster so fresh bundles reach every site.
- The heavy work runs in a Web Worker created from an in-memory blob: the main thread only captures and batches events, while the worker serializes protobuf and POSTs to the receiver without ever blocking the page.
- A session begins with `sid = 'NEW'`; the receiver answers with the real session id, which the recorder keeps in sessionStorage under the key `lognroll` for the life of the tab.
- `identifyUser` attaches a name and email to a session through a single `IDENTIFY` log point, applied once by the receiver so a session's identity cannot be overwritten.
- The client's core rule is that recording must never break the host: loading, storage, worker creation, console capture, and network sends all degrade to warnings rather than exceptions, so the page survives any recorder failure.

- What still goes wrong is configuration, not code: CSP policies that omit the LogNroll hosts or blob workers, ad blockers that treat recorder traffic like tracking, and a blocked `NEW` handshake — all diagnosable from the network tab and the `[LognRoll]` console warnings.

Chapter 5: Capturing a User's World

A browser tab is a machine that never stops talking. The user moves a pointer, presses a key, scrolls, clicks; behind the scenes the page mutates its DOM, loads stylesheets, fires network requests, and logs messages. If a session replay is the story of what a person did on a website, the recorder is the journalist who never blinks — noticing every moment that matters and writing it down in a form someone else, years later, can read back.

This chapter goes inside that journalist. We walk the recorder bundle (`logger.lnr.1.0.1.js`, whose TypeScript source lives in the LogNroll logger repository) and examine each event type it produces: what is captured, how, what the payload looks like, and what it costs. The running theme is a three-way trade: every capture choice trades **fidelity** (how perfectly a replay reconstructs reality) against **volume** (how many bytes must travel and be stored) against **performance** (how much of the host page's budget the recorder consumes). One constraint shapes everything: the recorder runs on somebody else's website, in the same thread that renders their checkout button.

The recorder produces fifteen kinds of events, named by a shared contract: `NAVIGATION`, `MUTATION`, `CLICK`, `INPUT`, `MOUSE_MOVE`, `LOG`, `NETWORK`, `SCROLL`, `KEYBOARD`, `FORM`, `META`, `IDENTIFY`, `STYLES`, `PING`, and `PERFORMANCE`. Each gets a section below, with payloads as the recorder actually builds them. Mouse movement and DOM mutation get extra attention, because they are where replay systems win or lose against volume and complexity; Chapter 6 covers the binary envelope all events ride in, and Chapter 7 covers the privacy rules that constrain capture.

The event pipeline at a glance

Before individual event types, see where every event goes. The recorder runs in two execution contexts: the **main thread**, where the page lives and every listener is attached, and a **Web Worker**, spawned from a blob of JavaScript at startup. Capture happens on the main thread; transport happens in the worker.

The seam between them is a queue. Every tracker hands the `LoggerAPI` an event object shaped like this:

```
{
  type: eventType,      // e.g. 'CLICK'
  timestamp: Date.now(),
  data: data,           // type-specific payload string
  index: this.index++   // per-event sequence number
}
```

Events accumulate in an in-memory set, and every 100 milliseconds (`batchDelay`) the main thread posts the whole set to the worker and clears it:

```
window['lnrwrk'].worker.postMessage({ type: 'event', data: this.eventQueue });
this.eventQueue.clear();
```

The worker encodes each event into a protobuf `LogPoint`, packs points into chunks (capped at roughly 1 MB), and POSTs them to the receiver — encoding, network I/O, retries, and backoff never touch the main thread. We return to why that split matters at the end of the chapter.

Every session opens with a short prologue. On boot the recorder emits a `NAVIGATION` event with the page URL, a `MUTATION` event containing a full serialization of the page's HTML (the **snapshot**), the initial scroll position, a `META` map of device and browser metadata, and `STYLES` events for the page's stylesheets.

Navigation: following the page through space and time

The simplest event is also the backbone of the session. A `NAVIGATION` event marks "the page the user sees now is at this URL," and its payload is nothing but the URL string:

```
this.api.sendEvent(CONFIG.EVENT_TYPES.NAVIGATION, window.location.href);
```

The recorder emits one at boot, then stays alert for two flavors of navigation. A full page load — a hard navigation, a typed address — ends the current session; the next page's recorder boots a new one. But modern sites are **single-page applications (SPAs)** that change "pages" without reloading. To follow those journeys, the recorder instruments the routing primitives: it wraps `history.pushState` and `history.replaceState`, listens for `popstate` (back and forward buttons) and `hashchange`, and on any URL change emits a fresh `NAVIGATION` event, re-checks dynamic styles, and re-sends the `META` map, since the new "page" may have a different title or document state.

Navigation capture costs almost nothing — one small event per route change — yet it gives every later event its URL context, which is how analytics can group events by the page that preceded them.

Mutations: recording the DOM itself

Navigation tells you where the user is; mutations tell you what the page looked like. Every pixel the user sees is a rendering of the document tree, so a replay must reconstruct that tree at any moment — in two phases: one full-page snapshot at the start, then a running record of every change.

A snapshot, then the delta

At boot, the recorder serializes the entire document — the `outerHTML` of the `<html>` element — and sends it as a single `MUTATION` event. That is the **snapshot**. The serialization is not naive: the recorder first walks the live page assigning every element a monotonically increasing `lnrId` (an integer address stored as a property on the node), then mirrors that numbering onto the serialized copy so later events can refer to elements by number. It rewrites the `<base>` tag to an absolute URL so the snapshot renders correctly wherever it is later hosted, and strips `sourceMappingURL` comments.

From then on, a `MutationObserver` watches the whole document — configured on `document.documentElement` with `childList`, `subtree`, `characterData`, and `attributes`, plus the old-value flags for text and attributes — so it sees every node added or removed anywhere in the tree, every text change, and every attribute change, together with its previous value. At startup the recorder drains the observer's pending buffer once (`takeRecords()`), so mutations that raced the observer's activation are not lost.

What a mutation record contains

Each observed mutation becomes its own event whose JSON payload mirrors the browser's `MutationRecord`, enriched for replay:

```
{
  ts: ts,                    // performance.now() at capture time
  type: mutation.type,       // 'childList' | 'attributes' | 'characterData'
  lnrId: target.lnrId,       // which element changed
  mutationIndex: this.mutationIndex,
  target: { tagName, id, className, xpath, styles },
  attributeName, oldValue, newValue,   // for attribute changes
  oldText, newText,                    // for text changes
  addedNodes: [...],                   // full recursive element data
  removedNodes: [...],                 // "lnrId>>>xpath" references
  nextSibling
}
```

For an added subtree the recorder walks it recursively, serializing each element into a rich record: tag name, `id`, `className`, `xpath`, its `lnrId`, its current `value` (for form controls), its parsed inline `style`, its attributes (with `lnr-id` folded in), a `"lnrId>>>xpath"` next-sibling reference, and its children, recursively. Removed nodes can no longer be walked, so they are recorded as `"lnrId>>>xpath"` references telling the player what disappeared and from where. A `WeakSet` of serialized nodes guarantees each node is fully captured only once; later mutations reference it by `lnrId`.

Two details matter. First, every mutation event carries an `order` value (`mutationIndex`, a counter that only grows): one observer callback can deliver many records at once, and `order` lets the player apply them in the exact sequence the browser reported. Second, when the

serializer meets an `INPUT`, `TEXTAREA`, or `SELECT` for the first time, it installs an instance-level wrapper on the element's `value` property, so that framework-written values also produce a form event.

Why mutations are the hard part

Mutations are the most demanding capture in a replay system, for three reasons.

First, **volume and cost**: a mutation event is the most expensive event the recorder produces. Serializing an added subtree means walking nodes, reading attributes, parsing inline styles, computing XPaths, and building nested JSON — all on the main thread, mid-page-render. A chat widget re-rendering a list or an infinite scroll can add hundreds of nodes at once. The recorder chooses fidelity: it does not coalesce or debounce mutations, because merged DOM changes would make the replay visibly skip frames. It relies on snapshot-plus-delta to keep steady-state cost low — most of a session's DOM work happens once at boot, and the running stream is usually a trickle of small attribute and text changes.

Second, **correctness through ordering**. DOM events are only meaningful relative to one another: apply "remove node X" before "add node X" and the page is wrong; misapply an attribute change and the replay silently diverges. Hence the `order` counter and precise `lnrId` references, and the player's rule (Chapter 16) of applying mutation and navigation events first, rebuilding the DOM skeleton before replaying the interactions on top.

Third, **what a recorder cannot see**. The observer attaches to the top document and the serializer walks the light DOM. Cross-origin iframes are unreadable: the same-origin policy blocks script access to another origin's content, so the recorder sees only the `<iframe>` element, never its contents. Shadow DOM internals are likewise outside the traversal. DOM-based capture reconstructs the document tree it can observe, and every replay system draws this same boundary.

Clicks: position plus identity

A click is a deliberate act and the single most valuable interaction event in replay. Its payload is a compact comma-separated string:

```
this.api.sendEvent(CONFIG.EVENT_TYPES.CLICK,
  `${event.clientX},${event.clientY},${target?.lnrId},${target?.tagName},${relX},${relY},$
{xpath}`
);
```

Reading left to right: the viewport coordinates `x` and `y`; the clicked element's per-session `lnrId`; its `tagName`; two relative offsets `relX` and `relY`; and the element's `xpath`. The offsets are the click position measured from the element's border-box corner (`clientX` minus `getBoundingClientRect().left`), rounded to pixels. The recorder's own comment explains

why: when a replay renders a click by resolving the element's XPath rather than absolute coordinates, the player's natural guess is the element's center — the stored offsets preserve the true click point through that translation, which keeps **heatmaps** honest.

The `lnrId` and `xpath` pair embodies identity across two timescales. `lnrId` is per-page-load numbering: precise within one session, meaningless across sessions, because numbering restarts at every boot. The XPath — computed by `getXPath`, which short-circuits to `//*[@id="..."]` for any element with an `id` and otherwise builds positional steps like `/html/body/div[3]/button[1]` — is a stable address across sessions, which is what makes cross-session aggregation possible: the heatmap processor keys clicks by XPath only, never by `lnrId`. Clicks are cheap and unthrottled.

Typing: keyboard, input, and form values

Text entry earns a recorder its keep for debugging — and demands the most care. The contract reserves three event types for it, at different granularities.

Keyboard events capture the raw keys. The recorder listens for `keypress`, `keydown`, and `keyup` on the document, skips pure modifier keys (`Shift`, `Control`, `Alt`, `Meta`, `CapsLock`, `Tab`), and batches behind a 100-millisecond timer (`BATCH_TIMEOUT`) so a burst of typing becomes a few events rather than one per keystroke. When the timer fires, the burst is compressed into a compact JSON array sent as one `KEYBOARD` event:

```
{ k: event.key, t: event.type, ts: event.timestamp,
  c: event.ctrlKey, a: event.altKey, s: event.shiftKey,
  m: event.metaKey, r: event.repeat,
  tgt: { tag: target.tagName, id: target.id, cls: target.className,
         xp: xpath, lnrId: target.lnrId } }
```

Keyboard events carry the actual key value plus modifier state and target identity — enough to replay shortcuts and spot a user hammering Enter. They do not carry field context.

Form events are the value-capture channel, operating on what a control *contains* rather than which keys produced it. This is the right abstraction for replay: pasted text, autofill, and framework-written values produce no keystrokes yet still change the form. The recorder listens for `input` events with a 500-millisecond debounce per element, `change` events, and `submit` events, which snapshot the entire form. Each event carries an element record and, where present, the containing form:

```
{
  type: 'input' | 'change' | 'submit',
  timestamp: Date.now(),
  element: { type, id, name, xpath, value, lnrId },
  form: { id, name, action, method, xpath, lnrId }
}
```

On submit, a `fields` array holds one record per non-excluded control — so a replay can show exactly what the user had typed when they pressed Pay. A map of last-known values suppresses repeats: an event whose value matches the last recorded one produces nothing.

And here is the caveat. Form capture is *value* capture, and values are what privacy rules protect. The recorder therefore applies a first line of defense, hard-coded in its configuration: `password` inputs are never captured; any field whose `name`, `id`, or class contains `cvv`, `ssn`, `credit`, or `card` is excluded; and values are truncated at 1,000 characters with an ellipsis. These rules are a baseline, not the whole story — the site owner's masking and sanitization policy lives in the SDK layer, and Chapter 7 covers that machinery. The rule of thumb: the recorder captures form values by default because replay without them is blind.

One distinction deserves emphasis. The contract reserves `INPUT` (enum 3) for fine-grained text-entry events — "a text input or value change in a form control." In the current recorder, value changes travel through the `FORM` channel and discrete keys through `KEYBOARD`; `INPUT` remains reserved for future capture strategies or other embedders. The lesson: the type enum is a stable public contract (Chapter 6), while which types a given recorder emits is an implementation choice.

Mouse movement: the volumetric enemy

Clicks are the most valuable events and mutations the most complex; **mouse movement** is the most dangerous — not because it is hard to record but because it is nearly infinite. Browsers deliver `mousemove` at display refresh rates, sixty or more events per second while the pointer moves; ten active minutes can yield tens of thousands of candidates.

The defense has three layers. First, a **distance gate**: a sample is kept only if it moved at least 1 pixel from the previous one (`MIN_DISTANCE`); a stationary pointer produces nothing. Second, samples accumulate in memory rather than becoming events immediately: each qualifying position `{x, y, timestamp}` is pushed onto an array and a 200-millisecond timer (`BATCH_TIMEOUT`) restarts. The timer is trailing — it fires only after the pointer rests, so an entire continuous sweep is collected into one burst. When it fires, the burst folds into a single compact string:

```
const formattedString = this.events
    .map(event => `${event.x},${event.y},${event.timestamp}`)
    .join('|');
```

One `MOUSE_MOVE` event carries the whole sweep, each sample as `x,y,timestamp` separated by a pipe. Third, that event rides the normal pipeline to the worker like any other.

The arithmetic matters (Chapter 19 builds cost models on it): the gate caps samples near one per pixel of travel, the 200-millisecond grouping amortizes per-event overhead across a sweep, and the flat encoding avoids JSON's repeated key names per point. Even so, mouse

movement remains the largest volume contributor on interaction-heavy pages, because it scales with motion rather than intent.

Scrolling: absolute position, anchored replay

Scroll is the page's answer to mousemove — continuous, and needing sampling without drowning. The recorder listens for `scroll` events on window and document in the capture phase and builds a payload answering two questions: where is the viewport now, and what is the user looking at.

```
{
  x, y,                    // absolute window scroll offsets
  maxX, maxY,              // scrollable extent minus viewport size
  target: {                // what scrolled
    element: 'window' | 'element',
    xpath, id, tagName, className,
    dimensions: { width, height, scrollWidth, scrollHeight }
  },
  relativeTarget: {        // first visible content element in the scroller
    lnrId, top, scrollTop
  } | null,
  elementScroll: {         // element-scroller geometry, when an element scrolls
    scrollTop, scrollLeft, scrollWidth, scrollHeight,
    clientWidth, clientHeight, maxScrollX, maxScrollY
  } | null
}
```

`x` and `y` are absolute offsets; `maxX` and `maxY` are the scrollable extent minus the viewport, so consumers can turn offsets into progress. When an inner element scrolls, `target` describes it and `elementScroll` carries its internal geometry: offsets, content size, and its own maxima.

The most interesting field is `relativeTarget`. Raw scroll offsets replay perfectly only if the replay viewport matches the recorded one — and it often does not, because the player runs in its own frame. So the recorder also finds the first visible content element inside the scrolled region — skipping the container itself, `<html>`, `<body>`, invisible and `position: fixed` elements, and anything smaller than 5 by 5 pixels — and reports its `lnrId`, its offset from the container top, and the container's `scrollTop`. During replay that element is an **anchor**: the player restores scroll by locating and aligning the anchored element, which survives viewport differences far better than a raw pixel offset. Chapter 16 returns to this trick in the player's scroll-sync logic.

Scroll payloads are a handful of numbers; the one expensive moment is the anchor search, which walks candidates and reads computed styles on every scroll event — the price of robust replay scroll.

Console messages: the voice of the page

The `console` is where a page confesses its problems, and a replay that cannot hear it misses the most useful debugging signal of all: the page's own errors. Console capture lives in the SDK layer around the recorder. The wrapper replaces `console.log`, `info`, `warn`, `error`, `debug`, and `trace` on `window` with functions that first call the original method — the page behaves exactly as before, the recorder never muffles a message — and then record it, level-prefixed, with arguments joined:

```
const data = method + "<|||>" + args.map(arg =>
    typeof arg === 'object' && arg !== null
        ? JSON.stringify(arg, circularReplacer)   // objects become JSON
        : String(arg)
).join('<SPLIT>');
```

A handler calling `console.error("payment failed", {code: 502})` becomes `error<|||>payment failed<SPLIT>{"code":502}` — the level recoverable from the prefix, objects surviving as JSON rather than `[object Object]`. A circular-replacer substitutes `[Circular]` for self-referencing objects, and capture is wrapped in `try/catch`: a capture failure must never break the page being recorded.

The wrapped methods push `{data, timestamp}` entries into a buffer; the recorder drains it once per second into `LOG` events. Because `LOG` events carry their level in the payload, the player can paint them like a developer console — gray for `log`, amber for `warn`, red for `error` — and badge sessions that contain errors. That is the feature that cracks the Candlewood Books mystery at the end of this chapter.

Network requests: watching the page talk

Every modern page is a conversation with dozens of servers, and many of the worst bugs live in it: a request that never returns, a 500 swallowed by a silent `catch`, a slow API freezing the UI. The recorder captures the conversation with the same pattern as console — the SDK wraps the browser's network primitives, the recorder drains the results.

Two wrappers are installed. `window.fetch` is replaced by a function that runs the real fetch but records the request — URL, method, body, headers — before it goes out, then records the response: status, `statusText`, headers, body text (read from a cloned response so the page's own read is unaffected), and wall-clock `responseTime`. `XMLHttpRequest` is wrapped by replacing the constructor: each instance's `open` and `setRequestHeader` are intercepted to remember URL, method, and headers, and `onreadystatechange` is chained so that at `DONE` the same recording happens. A third, quieter channel taps the browser's **resource timing** API: every second the wrapper sweeps `performance.getEntriesByType('resource')` for static assets

the wrappers never saw, records their timing, and clears the buffer. That is how a replay can show the hero image that stalled a mobile page even though the browser itself loaded it.

Every recorded exchange becomes a buffered log entry:

```
{
  type: 'network',
  stage: 'response' | 'error',
  payload: {
    request: { id, url, method, body, headers, initiatorType, timestamp },
    response: { url, status, statusText, headers, data, timestamp, responseTime },
    error: { message, timestamp }     // when the request failed outright
  }
}
```

Three hygiene rules keep noise and risk down. Requests whose URL contains `lognroll` are skipped — the recorder must not record its own traffic to the receiver. Captured requests and responses each pass through configurable **sanitizers**: a `requestSanitizer` returning `null` suppresses that request pair entirely; a `responseSanitizer` returning `null` redacts the response — body, headers, and data stripped, leaving URL, status, timing, and `responseTime` — so a team can see that a payment call failed without ever persisting the provider's payload. And sites may opt into **trace parameters**, tagging dynamic-content URLs (never static resources) with a request id (`?rid=`) or device id (`?sid=`) so a replayed request can be correlated with backend logs. Sanitizers get their full treatment in Chapter 7.

Performance signals: watching the page suffer

Replay answers "what happened"; performance capture answers "what did it feel like" — the jank and freezes that never appear as a console error. The recorder watches two things. Every sixty seconds, where the browser exposes it (Chrome-family browsers do), it reads `window.performance.memory` — the JavaScript heap's limit, total, and used sizes — and emits a `PERFORMANCE` event with the three numbers comma-separated:

```
'MEMORY,' + memory.jsHeapSizeLimit + ',' + memory.totalJSHeapSize + ',' + memory.usedJSHea
pSize
```

A climbing used heap across a session fingerprints a memory leak. The second feed is sharper: a `PerformanceObserver` watches for **long tasks** — tasks monopolizing the main thread past the ~50-millisecond threshold where users perceive a freeze — and each becomes a `PERFORMANCE` event:

```
`LONGTASK,${entry.name}: ${entry.startTime}ms`
```

Long tasks are the technical definition of "the site freezes on my phone": while the main thread is blocked, clicks queue and nothing happens until the task finishes. Correlating a long task with the replay timeline shows what the page was doing right before the freeze.

Page metadata: who is looking, through what window

Pixels are only interpretable if you know the glass they were drawn on. A `META` event carries a flat map of dozens of environment facts, sent at session start and refreshed on navigation: screen and window geometry (dimensions, available area, `colorDepth`, orientation, viewport, `devicePixelRatio`, computed zoom); browser and platform (`userAgent`, language, `cookieEnabled`, `doNotTrack`); capability flags (local and session storage, Web Workers, service workers, WebGL 1 and 2, touch, geolocation, notifications, Bluetooth, USB, NFC, battery, Permissions API, media devices, WebRTC, WebSocket); document state (`pageTitle`, `pageReferrer`, `pageDomain`, `pageURL`, protocol, `readyState`, character set, content type); and time and place (timestamp, `timezoneOffset`, IANA `timezone`).

Most of the map is one event of a few hundred bytes. Its value shows in the session list and in debugging: the “only happens on this device” bug is half-solved the moment you see a 320-pixel viewport, a 2× pixel ratio, and no service worker. The flags also document which capture capabilities the recorder could rely on in that session.

Stylesheets: making the replay look right

A faithful replay must reproduce not just the DOM but the styles that made it look the way the user saw it. The strategy mirrors snapshot-and-delta. At boot the recorder walks `<link rel="stylesheet">` elements and emits a `STYLES` event per stylesheet carrying its URL, which the player can fetch and apply. Then it watches the harder case: **dynamic styles**, the CSS-in-JS output injected into `<style>` tags at runtime, which has no URL and changes as the app re-renders. For those, the recorder patches the stylesheet API — wrapping `CSSStyleSheet.prototype.insertRule`, `addRule`, `replace`, and `deleteRule` — so any rule-level change emits a `STYLES` event naming the owning style tag and carrying the sheet’s full current text:

```
'STYLED>>>' + styleTag.lnrId + '>>>' + styleText
```

A framework adding a class’s rules moments before an element with that class appears thus produces a style event *before* the mutation that uses them — an ordering the player depends on. As a backstop, the recorder polls every second for style tags bearing the data attributes of common CSS-in-JS libraries (`data-styled`, `data-s`, `data-emotion`), hashes each sheet’s text, and emits whenever the hash changes — catching whole-sheet rewrites that bypass the patched methods. Cost scales with how dynamic a page’s styling is: theme re-injection on every route yields a steady trickle of full-sheet payloads; a statically styled site pays only the initial URL events.

Identity: who the session belongs to

The platform can record a session without knowing whose it is; anonymous visitors are perfectly replayable. But replay becomes more useful when a known user's session is findable by name and linkable across visits. The site owner declares identity through the SDK's `identifyUser(userId, traits)` call, recording a user id plus traits (in the reference SDK, `name` and `email`). The recorder watches for that declaration: on each queue flush it checks whether the SDK has set a user id, and the first time it sees one, it emits an `IDENTIFY` event with the identity as JSON:

```
{ id: lnr.userId, name: lnr.traits.name, email: lnr.traits.email }
```

An `identified` flag ensures this happens exactly once per session. One event, and the session is permanently associated with that person — searchable by name, linkable across the device chain, attributable when support sits down to watch. The privacy weight of that association is why Chapter 7 exists: identification turns an anonymous trace into a record about a person.

Heartbeats: proving the page is still alive

Most capture answers "what changed"; `PING` answers "is anything happening at all." A quiet tab — a user reading for five minutes — produces almost no events, and the platform cannot distinguish "reading" from "closed the tab" without a signal. `PING` is that heartbeat.

The contract reserves `PING` (enum 13), and the recorder's handling shows how real code drifts from its ideal shape. In the SDK-loaded recorder bundle the periodic ping timer is present but commented out — the recorder's own steady batch traffic was judged enough liveness signal in that revision. The embedded "full" bundle, for environments that preload the recorder directly, keeps an active ping: one `PING` event every 60 seconds of quiet. Between the two, liveness is a mixture of real heartbeats and the side effects of normal traffic. Chapter 8 picks up how the transport treats quiet sessions.

Ordering and timestamps: putting events in their place

A replay is only as good as the order of its events, and the recorder invests in three mechanisms to make ordering exact. The **timestamp** — `Date.now()` at capture, in milliseconds — anchors events to wall-clock time: it builds the timeline, buckets events into analytics minutes, and correlates a replay with server logs from the same moment. The **per-event index** stamps every event with a counter that only increments, so even two events created in the same millisecond have a definitive order; the recorder persists this counter to `sessionStorage` under `lnr-index`, and on page unload writes it forward by a thousand before flushing — headroom so a newly loaded page's events can never collide with the tail of the

page that just left. The third mechanism, `order`, belongs to mutations: records arrive from the browser in batches, and `order` preserves the browser's delivery sequence for the one event class where ordering is correctness, not chronology. Together they let any consumer reconstruct a definitive sequence — the foundation of the player's NAVIGATION/MUTATION-first rule in Chapter 16.

One subtlety shows the recorder thinking of sessions as containers, not just timelines. The session id lives in `sessionStorage`; a fresh tab starts with the sentinel `NEW`, and the receiver answers with a real id. If the recorder ever sees its session id *change* to a different real id mid-page — the platform deciding the visit belongs to a new session — it treats that as a new beginning and re-emits the whole prologue: a fresh snapshot and metadata under the new id, so the new session's replay starts from complete state. Events are cheap to re-baseline; an un-baselined replay is not.

Off the main thread: why the worker matters

The chapter opened with the pipeline's shape — capture on the main thread, transport in a worker — and the reasoning is a design principle, not convenience. The main thread is a scarce, shared resource: every millisecond the recorder spends on it is one the page cannot spend rendering or handling input. A recorder that serialized protobuf, tracked retries, and managed network state on the main thread would steal that budget in visible ways — the very jank replay exists to diagnose. So the recorder draws a hard line: the main thread does the cheap work (listening, formatting compact payload strings, queueing), handing raw events to the worker roughly every 100 milliseconds; the worker does the expensive and stateful work — protobuf encoding, chunking, POSTing, retrying with backoff.

The line even shows in the configuration handshake: when the recorder starts the worker it copies its configuration across but strips the network configuration first, because that config holds sanitizer *functions*, and functions cannot cross the thread boundary.

The deal is not free: `postMessage` has cost, and mutation serialization is inherently a main-thread DOM walk. But the recorder keeps every optional cost off the critical thread, and keeps the page's worst enemy — unbounded network work under a flaky connection — out of the render path. Chapter 8 follows the events across that boundary and down the wire.

> **Story checkpoint — Candlewood Books:** The session that cracked the Safari checkout bug is almost boring until it isn't. Priya scrubs the replay of the customer who reported "nothing happens when I press Pay." The customer scrolls, opens the cart, clicks Checkout, fills the form. Then, in the timeline, two rows sit side by side: a red LOG entry — an uncaught JavaScript error from the checkout's submit handler, thrown by a stale cached bundle still referencing an element the redesign had removed — and, 400 milliseconds later, the CLICK on the Pay button that "did nothing." The error and the dead click, adjacent in time, tell the whole story: the handler crashed before it ran, so no request went out, no spinner appeared, and the button silently swallowed the customer's intent. No console digging, no "please clear your cache." Marta copies the replay link into the ticket and closes it in one reply.

Chapter takeaways

- A session is a stream of fifteen event types; each capture trades fidelity against volume and against the host page's own performance budget.
- The recorder works in two phases — a full snapshot (DOM, scroll, metadata, styles) at boot, then incremental deltas — and re-baselines the snapshot whenever a session id changes mid-page.
- DOM mutations are the hard part: recursive main-thread serialization, order-sensitive application, and structural blindness to cross-origin iframes and shadow DOM.
- Mouse movement is the volumetric enemy: a 1-pixel distance gate and a 200-millisecond trailing batch fold cursor sweeps into single pipe-delimited events.
- Form capture is value capture, guarded at the recorder by hard-coded exclusions (password types, card-like field names, 1,000-character truncation); site-level masking policy lives in the SDK layer (Chapter 7).
- Identity, console, and network capture buffer their output and are wrapped so capture failures never break the page; protobuf encoding, chunking, and retries run in a Web Worker, off the page's render path.

Chapter 6: Protobuf: The Wire Language of Replay

Every event in a session — every click, keystroke, and DOM mutation — must leave the browser, cross the internet, be stored for weeks, and one day return to a player that reconstructs it. That journey only works if everyone along the path agrees on a common way to write events down. The previous chapter introduced the fifteen event types and their payloads; this chapter is about the agreement itself: the **protobuf** schema called `LogPoint.proto`, the binary format every event actually travels in, and the design decisions baked into its six fields.

The wire format deserves its own chapter because it is the one artifact every other component — recorder, receiver, worker, processor, player — shares, and because the choice quietly decides how cheap the whole system is to run. Each event is small — a few hundred bytes at most, often far less — but sessions hold many of them, they are produced continuously by every visitor, and each is encoded and decoded several times as it flows through the pipeline. Multiply a volume problem by a format problem and you get a cost problem; binary encoding is how replay systems keep that cost small.

The reference implementation uses **Protocol Buffers** — "protobuf" — Google's language-neutral mechanism for serializing structured data, defined by a `.proto` schema and compiled into native code per language. LogNroll's contract lives in `LogPoint.proto`, with identical copies in the logger, receiver, worker, session-processor, player API, and player frontend repositories. By the end of this chapter you will understand the schema field by field, be able to read an encoded event's bytes by hand, and know why protobuf fits a session-replay pipeline especially well.

Why binary beats JSON here

JSON is most developers' default answer to "how should two programs exchange data": readable, forgiving, supported everywhere. For a session replay pipeline it has one fatal flaw: it repeats itself. A minimal event as JSON spells the keys `timestamp`, `type`, and `data` in full on every line, and writes numbers in decimal ASCII — a 13-digit epoch timestamp costs 13 bytes before it encodes anything, where binary needs at most 8. Measured across a real session's event mix, a protobuf encoding of the same data is roughly **three to five times smaller** than the equivalent JSON. For a platform that stores petabytes of sessions, that ratio is the entire argument.

Size is only the first win. Binary formats parse faster — the decoder walks typed fields instead of tokenizing text, and allocates no strings for repeated key names — and smaller payloads

mean lower bandwidth and fewer round-trips under load. But the third advantage matters most to a long-lived system: **schema evolution**. A session recorded today may be replayed in two years by a player that did not exist when it was recorded. Protobuf is designed for that world. Fields are identified by numbers, not names, and a decoder simply skips any field number it does not recognize; new payloads and new fields coexist with old consumers, which ignore what they do not understand. JSON has no equivalent discipline.

Tags, not names

The trick that makes protobuf compact and evolution-safe is that fields on the wire are identified by **field numbers**, never names. The `.proto` assigns each field a number, and that number is the field's true identity: renaming a field in the schema does not change the bytes; renumbering one corrupts every message in flight. This is why the schema insists on its numbers 1 through 6.

Each value on the wire is prefixed by a **tag** that packs the field number together with its **wire type**, a hint telling the decoder how to read what follows:

```
tag = (field_number << 3) | wire_type
```

Wire types include `0` for varints (variable-length integers), `2` for length-delimited values (strings, bytes, embedded messages), and `5` for 32-bit fixed values. Because the tag carries the wire type, a decoder that does not recognize a field number can still skip its bytes correctly — the foundation of forward compatibility. Field numbers 1 through 15 are special: their tags fit in a single byte, which is exactly why the LogPoint envelope numbers its fields 1 through 6 with no gaps.

The LogPoint envelope

The whole contract is two messages: every event is a `LogPoint`, and a batch of events is a `LogPoints` holding repeated `LogPoint` items.

```
syntax = "proto3";

message LogPoint {
  int64 timestamp = 1;
  LogType type = 2;
  bytes data = 3;
  fixed32 version = 4;
  int64 order = 5;
  int64 index = 6;
}

message LogPoints {
  repeated LogPoint items = 1;
}
```

The `LogPoint` is an **envelope**: six fields identical for every event, whatever its kind. Any component can read any event's envelope without knowing anything about the event itself, then dispatch on `type`.

Field	Type	Meaning
`timestamp`	int64	Milliseconds since the Unix epoch when the event happened in the browser. The replay timeline is built from this value.
`type`	LogType	Which kind of event this is. The player and the processor dispatch on this value.
`data`	bytes	The type-specific payload. For most types it is a small JSON document; because it is `bytes`, payload formats can evolve without changing the envelope.
`version`	fixed32	Version of the payload or processing semantics for this event type. Consumers use it to interpret `data` correctly.
`order`	int64	Sequence number carried by DOM-mutation events, preserving the browser's delivery order for mutations — the one event class where applying events out of order visibly corrupts the replay.
`index`	int64	Per-event counter stamped by the recorder and persisted across page reloads; the chronological tiebreaker when timestamps collide.

Read the envelope as answers to questions every stage needs: `timestamp` answers "when," `type` answers "what kind," `data` answers "what exactly" in a form the envelope deliberately does not constrain, `version` answers "which dialect of `data`," and `order` and `index` answer "in what sequence" when timestamps tie or batches interleave.

The companion `LogType` enum names all fifteen kinds with explicit values — on the wire an enum is just an integer, and proto3 requires the numbers to be assigned rather than derived from name order:

```
enum LogType {
  NAVIGATION = 0;
  MUTATION = 1;
  CLICK = 2;
  INPUT = 3;
  MOUSE_MOVE = 4;
  LOG = 5;
  NETWORK = 6;
  SCROLL = 7;
  KEYBOARD = 8;
  FORM = 9;
  META = 10;
  IDENTIFY = 11;
  STYLES = 12;
  PING = 13;
  PERFORMANCE = 14;
}
```

Two properties matter. `NAVIGATION` is value 0, the proto3 default for an unset enum — an event whose `type` was never written decodes as `NAVIGATION`, not as garbage. And the numbers are append-only by convention: adding a type means taking the next number, never reusing an old one, because existing sessions were encoded with these values and will be decoded by software written years from now. The recorder's code refers to types by name and lets the generated bindings translate names to numbers; only the bytes know the truth.

The batch: `LogPoints`

The recorder never ships one event at a time. Recall from Chapter 5 that the main thread hands its queue to the worker roughly every 100 milliseconds, and the worker packs points into chunks of at most about 1 MB; each chunk is one `LogPoints` message, serialized and POSTed with content type `application/x-protobuf`.

The `repeated` field is where protobuf earns its keep on the transport leg. A repeated message field is length-delimited — each item is preceded by its byte length — so the encoder writes no separators beyond the lengths and the decoder can skip any item it cannot parse. Packing two hundred events into one message means the per-event overhead (envelope tags, HTTP headers, round-trip latency) is paid once per batch rather than once per event. Chapter 8 examines that arithmetic in detail.

Version, and why `data` is bytes

Two envelope fields embody the contract's strategy for surviving its own evolution.

`version` is a `fixed32` — always four bytes, little-endian — chosen deliberately over a varint, because a version number is read by every consumer of every event and constant width keeps it cheap to read and compare. The recorder currently stamps `version` 0 on every point; the field exists so that when a payload type's shape or meaning changes, the producer bumps the

number and consumers branch on it — render a version-1 payload this way, best-effort or skip a version-2 payload they have not learned. The same discipline appears server-side in the session processor, whose per-processor `PROCESS_VERSION` makes the pipeline skip sessions that already have the current version applied. The rule is a general law of replay systems: **bump the version whenever the shape or meaning of a payload changes, and make consumers gate on the version rather than assume.** Events are written once and read many times, often by software written later; `version` is the field that says the format is a contract with your future self.

`data` being `bytes` rather than a typed message looks like a cop-out, but it is deliberate, for three reasons. First, payloads are genuinely heterogeneous: a `MOUSE_MOVE` payload is a pipe-delimited run of coordinates, a `NAVIGATION` payload a bare URL string, a `MUTATION` payload a nested JSON document, a `META` payload a flat map of device facts. No single message shape fits them all without a giant union type or a schema so abstract it protects nothing. Second, because payloads live outside the envelope, each format evolves independently — a new click-payload version needs no envelope change, no new enum value, and no coordinated deploy across repositories. Third, `bytes` is length-delimited: the decoder reads a length and skips that many bytes, so an unknown payload written by a future recorder costs nothing to carry and nothing to ignore. The envelope is the stable spine; the payloads are the soft tissue that grows around it.

Order versus index in practice

Of the six fields, `order` and `index` are the easiest to gloss over and the easiest to get subtly wrong. The envelope deliberately does not dictate how they are stamped; the recorder's stamping makes their division of labor concrete. Every event receives an `index` from a counter that increments per event and is persisted across page reloads in `sessionStorage`, so even events created in the same millisecond — or on either side of a page unload — have a collision-free chronological order. Mutation events additionally carry `order`, stamped from the mutation counter that preserves the browser's delivery sequence for DOM records, the one class where applying events out of order produces a visibly wrong replay. In practice, then, `order` is where ordering is *correctness* and `index` is where ordering is *chronology*: a player that sorts by timestamp, breaks ties with `index`, and honors `order` for mutations gets a stream that is both time-true and structurally safe.

Reading the wire: a worked example

Binary formats are best understood by reading actual bytes, so let us encode one event by hand: a `NAVIGATION` event whose payload is the URL string `https://example.com`, with its timestamp simplified to 1,000 milliseconds for readability. (A real timestamp is the current

epoch in milliseconds — a much larger number — but the rules are identical; only the varint grows.)

Three fields get written, each beginning with its tag:

```
field   wire type   tag byte          value
1       varint(0)   0x08              timestamp 1000
2       varint(0)   0x10              type NAVIGATION = 0
3       bytes(2)    0x1A              length 0x13, then the URL's 19 bytes
```

Field 1, `timestamp`, has wire type 0 (varint): tag `(1 << 3) | 0` = `0x08`. Its value 1,000 is a **varint** — base-128, least-significant group first, every byte except the last carrying a continuation bit in its high position. One thousand in binary is `1111101000`; split into 7-bit groups from the right we get `0000111` and `1101000`, so the varint bytes are `0xE8 0x07`. Field 2, `type`, has tag `0x10` and value `NAVIGATION` = 0, a single `0x00`. Field 3, `data`, has wire type 2: tag `0x1A`, then the payload length 19 as the varint `0x13`, then the URL's 19 ASCII bytes verbatim.

The complete encoded message:

```
08 E8 07 10 00 1A 13 68 74 74 70 73 3A 2F 2F 65 78 61 6D 70 6C 65 2E 63 6F 6D
│  └────┘ │  │  │  └──────────────────────────────────────────────────────┘
│         │  │  │  └── 19 bytes of "https://example.com"
│         │  │  └───── payload length = 19 (0x13)
│         │  └──────── tag: field 3, wire type 2
│         └─────────── type = NAVIGATION (0)
└───────────────────── timestamp = 1000 (tag 0x08, varint 0xE8 0x07)
```

Twenty-six bytes for a complete, self-describing event including its URL — roughly half the equivalent JSON, and the gap widens as payloads grow and batches accumulate. Note what is absent from the bytes: field names, braces, quotes, any message boundary beyond the lengths. The schema is the decoder's memory; the bytes are pure data.

The example also shows why `int64` fields are varints while `version` is `fixed32`: timestamps and counters are usually small — a 13-digit epoch millisecond needs at most seven varint bytes, and tiny values need just one — so variable-length encoding saves space on the common case, while a hot, always-read field like `version` is cheaper at constant width. The trade cuts the other way too: `fixed32` costs 4 bytes even when its value is 0.

One JavaScript aside belongs here, because the recorder runs in a browser. JavaScript numbers are IEEE-754 doubles, safe for integers only up to 2^53 — comfortably above any epoch timestamp, but not a true 64-bit integer — so generated bindings represent `int64` through a wrapper (a long-like object or string) rather than a raw number, and decoding code converts explicitly. It is a small tax paid once in the decoder layer; the wire bytes themselves are identical regardless of which language produces or consumes them.

Protobuf in the browser, in Go, and in Java

Because protobuf is a definition compiled per language, the same `LogPoint.proto` yields different code across the system, all of it speaking the same bytes.

In the browser, the recorder's worker imports generated JavaScript bindings (`logpoint_pb.js` in the logger repository) and drives them imperatively: construct a `LogPoint`, call setters such as `setTimestamp` and `setType`, pack points into a `LogPoints`, then call `serializeBinary()` for the `Uint8Array` that `fetch` POSTs. The player frontend decodes the same bytes with its own generated module (`logpoint.ts`, from google-protobuf's TypeScript generator), reading mirror-image getters. Both sides are generated code; neither hand-writes a parser. Other browser bindings such as protobufjs are common in the ecosystem, and the wire format is what makes any of them interchangeable.

On the server the same schema compiles to Go and Java. The Go services — receiver, worker, player API — use the output of `protoc-gen-go`; the Java services, notably the session processor, use protobuf-java and honor the package options at the top of the file (`option java_package`, `option java_outer_classname`), which tell the generator which package and outer class to emit. Server-side bindings are strongly typed: `LogType` becomes a generated enum, repeated fields become slices or lists, and a mistyped field name fails at build time rather than at runtime. This is protobuf's core promise — one `.proto`, N languages, identical bytes — and it is exactly why a pipeline with a TypeScript recorder, Go ingestion services, and a Java analytics processor can share one contract without a custom parser anywhere.

One contract, many copies

There is, however, a wrinkle in how that shared contract is shared — and it is worth stating plainly because it is true. In the reference implementation, `LogPoint.proto` is **duplicated across repositories**: identical copies live in the logger, receiver, worker, session processor, player API, and player frontend, and the player checks in a generated TypeScript module beside its own copy of the `.proto`. There is no shared package, no single source of truth.

The duplication is real and acknowledged in the codebase, and it is best read as a deliberate trade made early by a small team: copying a 35-line schema file into each repository is nearly free, keeps services self-contained, and avoided the overhead of publishing a shared library when the platform was young. The cost arrives later, in the form of **lockstep**: adding a sixteenth event type means editing the `.proto` and regenerating bindings in six repositories at once, and the risk of drift grows with every change. The team's own notes flag this as a maintenance risk and point to the future direction — publishing the contract as one shared, versioned artifact — as the consolidation that will retire the copies (Chapter 20 returns to

that roadmap). For anyone building a replay system, the lesson is to decide early whether the contract lives in one shared artifact or in N copies, and to know which one you chose.

The duplication is survivable here partly because the contract is small and changes rarely — an event type is added a handful of times a year, not every sprint. That stability is itself a consequence of this chapter's design choices: the envelope rarely needs to change because payload evolution is absorbed by `version` and by `data` being `bytes`. The wire language was built to be boring, and in a system that must replay events recorded years apart, boring is a feature.

Chapter takeaways

- Protobuf encodes the same event stream roughly 3 to 5 times smaller than JSON, parses faster, and makes schema evolution safe — the three properties a high-volume, long-lived replay pipeline needs most.
- The `LogPoint` envelope (timestamp, type, data, version, order, index) is identical for all fifteen event types; payloads ride in `data` as untyped `bytes` so they evolve independently of the spine.
- Field numbers are a field's identity on the wire: the tag packs field number and wire type into one varint, and unknown fields are skipped by their length.
- `version` is the payload's dialect marker — bump it when a payload's shape or meaning changes and make consumers gate on it, as the processor does with `PROCESS_VERSION`; `data` as `bytes` keeps payload evolution out of the envelope.
- `order` preserves browser delivery sequence for mutations, where ordering is correctness, while `index` breaks timestamp ties across the whole session.
- A hand-encoded `NAVIGATION` event fits in 26 bytes, and the same `.proto` compiles to TypeScript, Go, and Java bindings that speak identical bytes — today deliberately duplicated per repository, with a single shared, versioned artifact on the roadmap.

Chapter 7: Privacy, Masking, and Consent on the Client

A session replay recorder is a machine that watches people. Chapter 5 showed you how thoroughly: it reads the rendered page, records what changes, remembers what is typed into form fields, and inspects the network requests your application makes. That thoroughness is what makes replay useful, and what makes it dangerous — the same event stream that lets a support agent see a failed checkout would also show them a customer's home address or password manager autofill. This chapter is about the controls that keep the useful part and cut the dangerous part: masking, sanitization, exclusion, and consent, applied on the client, where the data is born.

The principle behind everything here is simple to state and hard to practice: **data minimization at the source**. Capture the smallest amount of information that still does the job, and apply the rules in the visitor's browser, before anything is serialized onto the wire. If sensitive content never leaves the page, no encryption, access control, or deletion job downstream ever has to be perfect. As a LogNroll engineering article puts it, historical replays cannot be un-seen; masking and sampling belong in the launch plan, not in the post-mortem.

Why Replay Sees Everything — and Why That Is Dangerous

Replay records events, not pixels, but the events are rich. A **mutation event** carries the text content of nodes that change on the page, so whatever your application displays is what the recorder tends to capture: a welcome message with a customer's name, an order summary with a full shipping address, a dashboard showing account balances. A **form event** carries what was typed, not just where. A **network event** carries request and response bodies, and API traffic routinely includes tokens, personal data, and payment details. Even a page URL can be identifying: `/account/orders/48162` is a fact about a person.

None of this is a bug; it is the price of fidelity. To reconstruct what a user saw, the player needs to know what was on screen, and to diagnose "I typed my address and the form errored," you want the request and the response. But fidelity is indiscriminate: the recorder cannot tell that a text node holds a card number rather than a book title. That judgment is yours, and this chapter is about the tools for expressing it.

There is a second-order danger too: replay data is a durable record that attracts the wrong kind of attention — insiders with dashboard access, compromised accounts, subpoenas, scraper bots. Vendors in this market, LogNroll included, invest heavily in server-side access

control, encryption, retention, and abuse protection (Chapters 15 and 18), but the cheapest protection happens first: never collect the sensitive bytes at all.

Capture Only What You Need

Data minimization is the GDPR's word for a habit every engineer understands: don't store what you don't need. For replay, it means deciding which layers of the recording your use case actually requires before you record anything. Diagnosing a broken checkout needs clicks, form interactions, console errors, and network calls; it does not need a transcript of support-chat messages or of search suggestions.

LogNroll's recorder organizes capture into independent layers, each switchable from the init call. DOM recording (`dom.isEnabled`) controls the mutations that rebuild the page; switch it off and replay shows a blank page while console logs and network calls continue — enough when you only want error data. Network capture (`network.isEnabled`) can be disabled when only UI behavior matters. Each layer you keep is a layer you must audit for sensitive content; a common profile is full DOM capture on public pages, DOM with form masking on the storefront, and aggressive sanitization — or nothing at all — on account and admin screens.

Two more levers deserve mention. **Sampling** — recording a fraction of sessions — cuts the volume of personal data you hold without hiding bugs. **Retention limits** bound how long a session survives; LogNroll's removal job deletes archived sessions after thirty days by default (Chapter 12).

Masking Inputs at the Source

The sharpest privacy question in any replay deployment is also the most common: do we see what people type into fields? In a properly configured recorder the answer is no — sensitive input is **masked**, never recorded, or recorded in a form that cannot be read back — and the masking happens in the browser.

LogNroll's recorder ships conservative defaults for the fields it recognizes. Form tracking excludes inputs of type `password` outright and skips any field whose name, id, or CSS class contains `cvv`, `ssn`, `credit`, or `card`; long values are truncated at a thousand-character ceiling. The product documentation states the policy in plainer language: by default, password inputs are excluded, and sanitized elements render with zebra striping in playback so a viewer can see that content was withheld.

Beyond the defaults, masking is declarative and granular. Adding the `data-private` attribute to an element makes the recorder treat it and its children as off-limits; only their dimensions are recorded, so the layout survives. `data-private="delete"` hides an element from playback entirely, and `data-private="lipsum"` on an input or textarea records only the *shape* of the

typing — the length — so the player can show placeholder characters without ever capturing the characters themselves. A `data-public` attribute on a child overrides an ancestor's masking — an allowlist escape hatch: hide a whole form, then expose the fields that are safe.

Two recorder-wide switches apply the same idea without touching markup. `dom.inputSanitizer` obfuscates every user-input element on the page, and `dom.textSanitizer` obfuscates every text node; both honor `data-public` allowlists, and either can be set to the string `"lipsum"` for length-matched placeholders. The pattern to internalize: masking happens at capture time, so plaintext never enters the event queue, never serializes into protobuf, and never reaches the receiver or the archive. Once the bytes are gone, no later stage — not even the decryption key holders — can recover them.

Sanitizing Network Payloads

Form fields are only half the story. The recorder also wraps `fetch` and `XMLHttpRequest` and, by default, notes the method, URL, headers, and bodies of requests and responses (Chapter 5) — and HTTP is where the serious secrets live: `Authorization` headers, API keys in query strings, payment payloads, customer data. A **network sanitizer** is a function you supply that runs in the page on the real request and response objects before anything is queued. It can modify; it can also veto.

```
LognRoll.initSession(YOUR_APP_ID, {
  network: {
    // Strip credentials before a request is recorded.
    requestSanitizer: (request) => {
      if (request.url.includes("/api")) {
        return {
          ...request,
          headers: { ...request.headers, Authorization: "" },
        };
      }
      return request;
    },
    // Keep only timing and status for responses that may hold personal data.
    responseSanitizer: (response) => {
      if (response.url.includes("/account")) {
        return null; // redacts everything except timing data
      }
      return response;
    },
  },
});
```

Two details matter about this API. Returning `null` from a request sanitizer drops that request/response pair from the recording while the network call still happens normally — the wrapper never interferes with the page's own traffic. Returning `null` from a response sanitizer keeps the URL, status, and timing but removes bodies and headers, which is useful when you want to debug slow or failing calls without preserving their content. A common

pattern: null out `Authorization` or `x-auth-token` on the way out; on the way back, delete a field outright or replace its value with a placeholder like `**redacted**`; reserve full capture for endpoints you have audited.

URLs need their own treatment because they are recorded even for requests whose bodies you stripped. LogNroll's `urlSanitizer`, configured under `browser`, rewrites a URL before it is stored — the docs demonstrate redacting a path segment such as `/ssn/123-45-6789` and scrubbing query parameters like `secret_key`. A token in a query string can outlive the request that carried it.

Finally, note where sanitizers run. The network config holds functions, and functions cannot cross the Web Worker boundary, so LogNroll strips that config before handing state to its worker (Chapter 4). Sanitization happens on the main thread, at interception time. A sanitizer is a point of trust, not a post-processing filter: it sees what your application code sees, and it deserves the same review and tests as any code that touches customer data.

Selector-Based Exclusion

Attributes and blocklists give you a third way to think about masking, as a **selector-based exclusion** system: the recorder consults your declarations before it records anything. In addition to `data-private` on individual elements, the configuration accepts two blocklists. `privateAttributeBlocklist` names attributes (such as `data-hide-this`) whose matching elements are treated as `data-private`, and `privateClassNameBlocklist` does the same for CSS class names. A finer-grained option, `hiddenAttributes`, removes specific HTML attributes — name and value — from a recorded element without hiding the element itself.

Declarations scale better than one-off edits. A bookstore that marks every element carrying customer identity with a shared class can cover a whole screen with one blocklist entry instead of auditing each redesign. That works only with naming discipline: if privacy depends on a convention such as class `js-private`, it belongs in the design system and the code-review checklist, so new templates cannot quietly add sensitive fields without the marker. Masking is a contract between the frontend team and the recorder, and like all contracts it needs tests: fill every sensitive field with recognizable dummy values on staging, watch the recording back, and hunt for anything that should not be there.

Consent and the GDPR Frame

LogNroll is a European product, and its privacy posture is shaped by European law, so a chapter on client-side privacy needs the consent picture. The GDPR is a set of obligations that apply whenever you process personal data, and session replays usually contain personal data, often trivially identifiable since the recorder can be told who the user is. For a team

turning on replay, the relevant pieces are: a **lawful basis** for the processing; **data minimization** (this chapter) and **storage limitation** (Chapter 12); and transparency, access, and deletion for data subjects. A replay platform's session lifecycle, ending in a `REMOVED` status that deletes the archived data, is what deletion requests ultimately lean on.

The lawful-basis choice is yours to make with counsel, but its shape matters. For recordings that include identifiable activity, many sites rely on consent obtained before the recorder starts; that is also the safer reading of the EU ePrivacy rules for non-essential tracking scripts, which is why Candlewood's story includes a consent note for visitors. Others argue **legitimate interest** for narrowly scoped, anonymized diagnostics. What is not defensible is recording everything by default and hoping nobody asks. Two product features make these choices real: an IP-capture toggle, because the docs note that disabling IP capture may be required under certain privacy regimes, and `dom.disablePageTitles`, because `document.title` is captured by default and can leak sensitive context into session metadata.

Replay changes your relationship with users from "we saw an error" to "we watched you." That is heavier: disclose it in your privacy policy in plain language, offer a real off switch where consent applies, and agree internally on who may open a replay and why. This is not legal advice — a deployment that touches payment data deserves a lawyer's review — but the controls in this chapter are what make the legal promises true.

What the LogNroll Privacy Configuration Offers

The documentation organizes privacy configuration around five surfaces, which is a good map of what a mature replay client should let you control:

- **DOM sanitization.** Element-level exclusion via `data-private` (with `delete` and `lipsum` modes and `data-public` allowlists); global `dom.inputSanitizer` and `dom.textSanitizer`; attribute and class blocklists; per-attribute removal via `hiddenAttributes`; and `dom.isEnabled` to stop DOM recording entirely. Sanitized elements render with zebra striping in the player.
- **Network sanitization.** A `requestSanitizer` that can ignore a request/response pair or scrub it; a `responseSanitizer` that can strip bodies, redact fields, or reduce a response to timing alone; and `network.isEnabled` as the kill switch. The sanitizer functions receive typed `NetworkRequest` and `NetworkResponse` objects, importable from `@lognroll/lib`.
- **URL sanitization.** A `urlSanitizer` function that rewrites recorded URLs, with documented patterns for path segments and query parameters.
- **IP capture.** A documented toggle for whether client IP addresses are captured for web sessions.
- **Page titles.** `dom.disablePageTitles` to stop capturing `document.title`.

Notice what is absent from that list: no server-side “redact after the fact” feature is sold as the solution, and no certification is advertised. The configuration is entirely about deciding, in the visitor’s browser, what will and will not be collected. For a tool whose job is watching people, the recording itself is the privacy boundary; encryption, access control, and retention are a second line of defense, not a substitute.

A Checklist for Turning On Replay

If you are the engineer putting replay in front of real users, here is a pragmatic sequence distilled from this chapter and from the failure modes the LogNroll team has seen:

1. **Map the sensitive surfaces** before writing any config: checkout and payment, account settings, admin and support tools, search, chat, and any page that renders personal data.
2. **Choose defaults that err on the side of hiding**, and make masking part of the initial implementation, not a follow-up ticket.
3. **Mask inputs at capture time**: password fields are excluded by default; add `data-private` or input sanitization for card fields, CVV-style fields, and anything the denylist does not cover.
4. **Sanitize network traffic**: strip `Authorization` and token headers, redact or drop bodies that carry personal data, and run a URL sanitizer for query-string secrets.
5. **Decide identity and IP policy**: only call `identify` when a session is meant to be tied to a person, and set the IP-capture toggle deliberately.
6. **Pick a retention window** that matches your incident response needs; thirty days comfortably covers a launch or a holiday season.
7. **Handle consent and disclosure**: add the consent note where EU rules require it, and describe the recording in your privacy policy in plain language.
8. **Test the masking** with a full dummy checkout on staging: type real-looking values into every field, submit, then watch the recording and confirm none of them appear.
9. **Treat the config as code**: keep it in the repository, review changes to it, and re-run the staging pass after every checkout redesign.
10. **Apply the grandmother test**: record people the way you would record your grandmother. If a session were your own — your address, your card, your search history — would you be comfortable with how it is recorded, stored, and who can open it? If not, the configuration is not done.

> **Story checkpoint — Candlewood Books:** Before recording went live on the storefront, Priya adds `data-private` to the checkout's card fields, turns on input sanitization, and strips payment request bodies from network capture. In the review meeting Tom asks, "wait, do we see card numbers?" Priya opens a staging recording of a full dummy checkout: the fields show as masked placeholders, the network panel shows the requests without bodies, and the answer is a simple, verifiable no.

Chapter takeaways

- A session replay recorder captures what the page displays, what users type, and what the network carries; that fidelity is the product, and also the privacy risk.
- The effective privacy boundary is the client: content masked or excluded in the browser never enters the event stream, so no downstream control ever needs to be perfect.
- LogNroll excludes password inputs and card-hinting fields by default, and extends masking through `data-private`, input and text sanitizers, and attribute and class blocklists.
- Network payloads are scrubbed by sanitizer functions that can ignore a request, strip headers and bodies, redact fields, or keep only timing data; URLs get their own sanitizer.
- GDPR-shaped practice means a lawful basis, minimization, bounded retention, and honest disclosure; identity and IP capture should be deliberate choices, not defaults.
- Turning on replay safely is a checklist exercise: map, mask, sanitize, test with dummy data, and treat the configuration as reviewed production code.

Chapter 8: Getting the Data Out: Batching and Transport

By the end of Chapter 5, a visitor's browser has turned their behavior into a stream of events, and by the end of Chapter 6 those events have a compact binary shape. None of that matters until the bytes reach the platform, and reaching the platform means crossing the most hostile part of the pipeline: the public internet, from a browser you do not control, over a connection that can drop mid-request, on a page that can close at any instant. This chapter is about how a recorder gets data out — the batching that makes it cheap, the HTTP contract that carries it, and the compromises that keep a session recording lossy-but-good rather than perfect-but-broken. The design goals, in priority order: never interfere with the user's page, never lose more than a sliver of a session, and keep the receiver's load reasonable. Everything in this chapter follows from those three goals.

The Shape of a POST

Events leave the browser as HTTP POSTs to the ingestion endpoint `POST /lognroll/{companyId}/{sid}` on the receiver host (in production, `receiver.lognroll.com`; the base URL is baked in at build time so each environment can point its recorder at the matching receiver). The **companyId** is the site owner's project key, embedded by the SDK at init. The **sid** is the session id, and as you will see shortly it can briefly be the literal string `NEW`. The body is a serialized `LogPoints` protobuf message — the repeated `LogPoint` records from Chapter 6 — sent with a `Content-Type` of `application/x-protobuf`. The recorder adds an `X-LogNroll-Device-Id` header when it has a device id, and the browser itself attaches the standard `User-Agent`, which the receiver parses to classify the device. The worker builds the request like this:

```
const headers: Record<string, string> = {
  'Content-Type': 'application/x-protobuf',
};
if (config.deviceId) {
  headers['X-LogNroll-Device-Id'] = config.deviceId;
}

const url = `${config.baseUrl}/lognroll/${config.companyId}/${config.sid}`;
```

The response is the session-id handshake: for a new session, its body is the freshly assigned session id as plain text, which the recorder stores. The receiver's CORS middleware answers every response — including its own errors — with permissive headers, because a response the browser cannot read is indistinguishable from a dropped connection (Chapter 9 returns to

why that matters for retries). This POST is the only network call the recorder makes: one endpoint, one method, one content type.

From Event to Batch: Two Buffers

Sending each captured event as its own request would be absurd: hundreds of requests per minute per visitor, each carrying tens of bytes plus a full HTTP round trip. Instead the recorder **batches**, in two stages that live on opposite sides of a Web Worker boundary.

On the main thread, every tracker hands its output to a shared event queue, a JavaScript `Set` of pending events. A timer that fires every 100 milliseconds (the `batchDelay`) hands whatever is queued to the worker and clears the queue, so an event waits at most about a tenth of a second before leaving the page. Some trackers pre-group before they reach the queue: mouse movements accumulate in a buffer flushed as a single event after 200 milliseconds of quiet (`BATCH_TIMEOUT`), so a sweep across the page becomes one `MOUSE_MOVE` event carrying a string of `x,y,timestamp` samples rather than hundreds of tiny records. Keyboard events batch the same way on a 100-millisecond timer, and form input is debounced at 500 milliseconds. The declared batch ceiling is `batchSize = 200` events — a cap that matters during bursts — while `batchDelay = 100` milliseconds sets the worst-case latency between an event and its departure.

The worker is the second buffer. It receives the queued events, serializes each into a `LogPoint` protobuf record, and packs them into upload chunks with a hard size limit of one megabyte — big enough that a pathological burst becomes a few large requests rather than thousands. Chunks wait in a first-in, first-out queue and are posted one at a time. Two consequences follow. The main thread never serializes and never does network I/O; its only job is capture and hand-off, which is the foundation of the “never block the checkout” rule at the end of this chapter. And ordering is preserved by construction: events carry an `index` assigned as they are created, chunks leave the worker in order, and only one POST is in flight at a time.

The NEW to Session-Id Handshake

A session needs an id before its events can be filed anywhere, but the recorder cannot mint one itself — ids are assigned centrally so the receiver, the archive, and the player agree on what belongs together. The contract is a small bootstrap dance. When a tab starts recording there is no session yet, so the recorder reads `sessionStorage` under the key `lognroll`, finds nothing, and adopts the placeholder sid `'NEW'`. On startup the worker POSTs an empty body to `/lognroll/{companyId}/NEW`; the receiver sees `NEW`, creates a session record, and answers with the real session id as plain text. The worker adopts that id, tells the main thread, and the

main thread writes it back into `sessionStorage` under `lognroll` — so if the user navigates or reloads, the same tab continues the same session instead of starting another.

The handshake itself retries with backoff a few times before giving up, because the receiver may be briefly unavailable at page load. The elegant part is that it does not need to succeed for recording to proceed: if it fails, subsequent batches are simply POSTed to `.../NEW`, and every successful POST returns the assigned session id in its response body, which the worker adopts on the spot. The session heals itself on the first batch that gets through, which makes the startup handshake an optimization rather than a single point of failure. Session continuity has one more piece: the event index counter is persisted across page unloads under the key `lnr-index`, so a session spanning several page loads keeps numbering events without collisions.

Retry, Backoff, and the Offline Reality

Network failures are a fact of life for a recorder, so the worker implements a bounded retry policy with real constants:

```
const MAX_SEND_ATTEMPTS = 5;
const RETRY_BASE_DELAY_MS = 1000;    // doubles per attempt
const RETRY_MAX_DELAY_MS = 30000;    // cap after doubling
const RETRY_SCHEDULE_MS = 3000;      // retry even when the user goes idle
```

Failed chunks stay at the head of the FIFO queue and are retried with **exponential backoff**: one second, then two, four, eight, up to thirty, with small random jitter so thousands of sessions recovering from an outage do not retry in lockstep. A timer keeps retrying in the background even when the user is idle, because a batch should not have to wait for the next click to discover the network recovered. The worker classifies failures: retriable HTTP errors (408, 425, 429, any 5xx) are retried; a non-retriable 4xx means the receiver understood and rejected the request, so the chunk is dropped after one attempt; and a network-level failure — `fetch` threw before a readable response — is retriable. A subtlety is worth understanding here: when the CDN in front of the receiver answers with one of its own error pages, that page carries no CORS headers, so the browser rejects it and the recorder sees a generic “Failed to fetch,” indistinguishable from a dead connection. Treating it as retriable is the right guess, because the batch is usually sendable once the origin recovers.

What happens when a chunk exhausts its five attempts? It is dropped, and dropped loudly: the worker logs a console error and reports the failure, so an operator sees the reason instead of silence. The recorder does not persist queued chunks to `localStorage` or IndexedDB, and it does not watch the network state; its offline behavior is the retry loop itself. That is deliberate simplicity — a durable outbox would need storage quotas and a second transport path for data that is diagnostic at the end of the day. The tradeoff, stated plainly: a chunk still waiting when the tab closes is lost. The recorder keeps only a few seconds of activity in

memory, retries hard while the page is open, and accepts losing the last moments before a close. For a debugging tool that is a good bargain; for a system that must not lose data, it would be a design flaw.

The Best-Effort Goodbye: Page-Unload Delivery

The hardest moment for any recorder is the instant the page goes away: a navigation, a closed tab, a crash. The classic remedies are `navigator.sendBeacon` and the `keepalive` flag on `fetch`, both of which ask the browser to finish a small request even as the page is torn down. It is worth being precise about what the LogNroll recorder actually does, because it does neither. Its unload handling is a `beforeunload` listener that persists the current event index and hands the queued events to the worker one last time.

That flush is best-effort in the truest sense. The worker's POST is an ordinary `fetch` without `keepalive`, and when a document is unloaded the browser is free to tear down the worker and cancel its in-flight request, so the final flush can be lost. The design compensates twice. First, the flush cadence is so short — 100 milliseconds — that the unload window can only hold a sliver of activity. Second, continuity lives in `sessionStorage`: the sid and the event index survive the navigation, so the next page load continues the same session, and the loss is a small gap rather than a broken session. A `sendBeacon` goodbye would close that gap a little more, and it is a fair improvement to ask for; the point is that the architecture already treats unload loss as a bounded, acceptable cost.

PING and the Liveness Question

A recorder that sends nothing during a long, quiet reading session looks dead to the server: no traffic, no updates to "last seen." The event catalog includes a purpose-built answer — the **PING** event type — and the classic recorder bundle emits one on a slow heartbeat interval (roughly once a minute) so that even an idle session ticks its liveness forward. The current worker-based bundle leaves that periodic PING commented out, an instructive honesty point about real systems: the interval was removed because the recorder already POSTs a batch every 100 milliseconds whenever anything is happening, and the receiver tracks first-seen and last-seen timestamps from whatever traffic arrives. For an active page, regular batches make a separate heartbeat redundant; for a truly idle page, a heartbeat only confirms the tab is still open, which is rarely worth a request. The PING type remains in the protocol for clients that want it.

When a Batch Is Lost: Best Effort versus Exactly-Once

Take stock of every place a batch can fail: the queue lives in memory, so a closed tab loses it; a chunk exhausts its five attempts and is dropped; a permanent 4xx discards it immediately. The transport, in other words, is **best effort** — with a loud failure mode instead of a silent one. Nothing in the pipeline tries to be **exactly-once**, and it is worth understanding why that is a choice rather than an omission.

Exactly-once delivery is among the most expensive promises in distributed systems. It requires every chunk to carry a unique id, receiver-side deduplication against stored state, and careful reasoning about retries after the server processed a request but its response was lost. LogNroll's transport makes the opposite bet: it is **at-least-once** in spirit, retrying until success or exhaustion, with no dedupe id on a chunk. The rare retry-after-lost-response can therefore deliver a duplicate batch, and the system tolerates it, because a duplicated event in a replay is a near-invisible artifact — the player applies the same mutation or paints the same mouse move twice — while a *lost* batch is a visible gap in the story. When data feeds precise counts, duplicates are bugs; when it feeds a reconstruction of what happened, duplicates are noise and gaps are the real cost. Batching shapes the blast radius too: a whole batch is at risk together, but a batch is at most a fraction of a second of activity or a megabyte of payload, so any single loss is bounded and local.

This is not a flaw to apologize for; it is the correct engineering answer for replay, and why the industry lands in the same place. The recorder keeps trying while the page is open, and the platform downstream keeps the session recoverable even when pieces are missing.

The Math of Small Batches

Batching is a latency-versus-throughput bargain, and the constants make the terms concrete. The table below is illustrative — real payload sizes depend on what the page does — but the numbers follow from the recorder's real constants.

When a POST fires	Typical events per POST	Added latency	Illustrative payload
Queue timer fires (every 100 ms) with events pending	1–10 during normal browsing	Up to 100 ms	Hundreds of bytes to a few KB
Mouse buffer flushes after 200 ms of quiet	One `MOUSE_MOVE` event holding several coordinate samples	Up to 200 ms	Tens to hundreds of bytes
Queue reaches the 200-event declared cap	Around 200	None extra (rare)	Tens of KB, payload-dependent

When a POST fires	Typical events per POST	Added latency	Illustrative payload
Chunk reaches the 1 MB worker cap	Hundreds to thousands	None extra (burst)	Up to 1 MB

Three conclusions fall out. First, most POSTs are tiny: the 100-millisecond cadence dominates, a normal page produces a steady trickle of small requests, and HTTP connection reuse means the per-request overhead is mostly headers. Second, the caps bound the worst case — without them, a mutation storm could produce a request large enough to strain the receiver. Third, the added latency is imperceptible to the person being recorded: every event is on its way within a tenth of a second, and nothing waits for a batch to fill. A design that filled large batches before transmitting would amortize overhead better, but it would add seconds of latency and push the unload-loss window from a sliver to a chunk. The recorder chooses small and often.

The Rule: Never Block the User's Checkout

Every mechanism in this chapter serves one rule, worth stating plainly: the recorder must never slow down the page it is watching, especially not at the worst possible moment, a checkout. If recording could stall a payment, no amount of debugging value would justify it.

The architecture enforces this structurally rather than by hope. Capture listeners are passive and cheap; they write small objects into an in-memory queue. The expensive work — protobuf serialization, chunk packing, HTTP, retry timers — runs in a Web Worker, off the main thread, so a slow or failing network never blocks rendering, input, or scripts. There is no synchronous request anywhere in the send path, and the recorder never waits on the receiver: a POST either succeeds, is retried in the background, or is dropped without ever touching the user's interaction. Even the capture wrappers fail safe — a sanitizer that throws logs a warning and lets the original request through, and a page that blocks the recorder's script gets no recording rather than a broken site.

It is also worth remembering what the recorder does *not* do, because "never block the checkout" gets confused in practice. The recorder's own POSTs are background traffic; they do not participate in the checkout flow. When the replay of a double charge shows two identical payment requests, those are the store's own requests, captured as network events — the recorder did not create them, it kept the receipt. Making the transport fast, batched, and loss-tolerant is what lets a replay service promise to be invisible; the store's own code still has to get the checkout right, and Chapter 16 shows how the replay exposes it when it does not.

> **Story checkpoint — Candlewood Books:** In the double-charge replay, Tom scrubs the network timeline and sees it immediately: two identical POSTs to the payments endpoint, 400 milliseconds apart, both fired from the same checkout. The payment gateway was slow, the Pay button never disabled, and the customer clicked again. The recorder did not create the duplicates — it captured each request as it fired and kept the receipt that proved the double submit.

Chapter takeaways

- Events leave the browser as protobuf POSTs to `/lognroll/{companyId}/{sid}` with a device-id header and the browser's User-Agent; the response body can carry the assigned session id.
- Batching is two-stage: a 100 ms main-thread flush with a 200-event ceiling, then a Web Worker that serializes and posts chunks capped at 1 MB, one request in flight at a time.
- The session starts with sid `'NEW'` and adopts the id the receiver returns; if the startup handshake fails, ordinary batches self-heal the session on their first successful POST.
- Retries are bounded and honest: up to five attempts with exponential backoff, distinct retriable and permanent failure classes, loud drops after exhaustion, and no durable offline queue.
- Unload delivery is best-effort by design — no beacon or keepalive — so losses are bounded to a sliver of activity, with session continuity kept in `sessionStorage`.
- The transport is best-effort rather than exactly-once on purpose: for replay, duplicated events are noise and gaps are the real cost, and small, frequent batches keep both small.

Chapter 9: The Receiver: First Touch of Every Event

Every session LogNroll ever replays enters the platform through one door, deliberately the least glamorous in the building. The receiver does not reconstruct pages, draw heatmaps, or detect errors — later services do all of that. It takes the protobuf batches a browser recorder POSTs over the public internet, checks that they belong to a real company and a real session, and pushes them deeper into the pipeline as fast as it can. In the reference implementation that door is a small Go service (the rewrite of an older Java receiver) answering at `receiver.lognroll.com`, running in Kubernetes behind nginx ingress and Cloudflare, and talking to exactly two other pieces of infrastructure: the shared MongoDB that holds session records and the NATS JetStream bus that carries event data onward.

The receiver is where almost every property of a session replay service is decided under time pressure. The browser is waiting on the HTTP response, so the work must be cheap; the request arrives from the open web, so it may be broken or hostile; and if the receiver drops an event, no worker or processor will ever see it. This chapter walks one POST from arrival to response, then examines the deliberate limits that keep the hot path thin; Chapter 10 takes over at the handoff, when a log point waits on the bus to be archived.

One Route, One Job

The receiver exposes a single interesting route, mirroring the recording contract from Chapter 8 exactly: the recorder sends `POST /lognroll/{companyId}/{id}` with a protobuf body. The first path segment identifies the customer company whose site the user is on. The second is the session id the recorder already holds — or the literal value `NEW`, when this is the first batch of a brand-new session and no id has been assigned yet.

Everything else hangs off that one route. The router uses the chi library, and its setup reads like a table of contents for this chapter:

```
r := chi.NewRouter()

r.Use(middleware.RealIP)
r.Use(panicRecoverer)          // structured panic log, CORS-carrying 500
r.Use(middleware.Heartbeat("/health"))
r.Use(corsMiddleware)          // CORS headers on every response
r.Use(httplog.RequestLogger(services.Logger, []string{"/health"}))
r.Use(diagnosticsMiddleware)   // ERROR on >= 500, WARN on slow requests
r.Use(noIndexMiddleware)       // X-Robots-Tag: noindex, nofollow
r.Use(blockCrawlersMiddleware) // 403 for known AI crawlers

r.Route("/lognroll", func(r chi.Router) {
    r.Post("/{companyId}/{id}",
        http.TimeoutHandler(
```

```
            http.HandlerFunc(services.handleLogPoints),
            requestTimeout, // 20 seconds
            "receiver-go: request timed out after 20s\n",
        ).ServeHTTP,
    )
})
```

The body is a serialized `LogPoints` batch container holding up to a couple of hundred `LogPoint` items. Alongside it ride three pieces of context the receiver treats as first-class: the `User-Agent` header, an `X-LogNroll-Device-Id` header (a client-generated id for grouping the sessions one user opens from a device over time), and the client IP, taken not from the socket but from `CF-Connecting-IP`, which Cloudflare sets at the edge — any other source would show the edge server's address instead of the visitor's.

Notice what the request does *not* carry: no API key, no signed token. The ingest endpoint is deliberately unauthenticated per request; it cannot be otherwise, because the recorder runs on customers' websites in browsers the platform does not control, and credentials embedded in page scripts would leak to anyone who read them. The tenant key is the company id in the URL itself, masking happens on the client (Chapter 7), and the events become meaningful to a logged-in dashboard user only later, behind the app's real authentication.

The Middleware Stack

Chi runs middleware in registration order, outermost first, so every request passes through the whole chain before the handler sees it and every response passes back out through it. Each layer earns its place.

Real client IP first. `middleware.RealIP` rewrites the remote address from the `X-Forwarded-For` chain maintained by nginx ingress. The handler then trusts `CF-Connecting-IP` when present — set at the edge, not spoofable by the caller — and otherwise strips the port from the socket address. The IP is stored on the session, feeds geolocation, and opens abuse investigations, so getting it right at the start matters.

Panic recovery with a CORS-carrying 500. Any handler can panic. The receiver replaces chi's stock recoverer with one that logs the panic as a structured JSON line with stack and request context, then returns HTTP 500 through the normal response path. Because the CORS middleware has already stamped the response by the time a panic unwinds, the browser sees a clean, readable 500, never a mysterious network failure.

Heartbeat. A built-in middleware answers `GET /health` with 200 and no database touched. Kubernetes probes and the platform's own operator tooling both check this URL. The health path is quieted in the access log so probes do not drown out real traffic.

CORS. Replay is cross-origin by definition: a recorder on `shop.candlewood.example` posts to `receiver.lognroll.com`. The CORS middleware allows all origins and the methods and

headers the recorder uses — including `X-LogNroll-Device-Id` — and answers `OPTIONS` preflights with 200 before application code runs. Because it wraps the whole router, *every* response, errors and timeouts included, carries the CORS headers. That invariant backs the timeout story below: nothing this service returns may ever look like a CORS failure to a browser.

Structured request logging. Every request becomes one JSON line via `httplog`, machine-readable and queryable, while the boot log marks each process start with its Kubernetes pod name. A browser-side "CORS exception" report can therefore be correlated with gaps in the access log: a crash or rolling deploy shows up as a bounded silence between boot markers.

Diagnostics. The innermost middleware records what the chain actually did — status, bytes, elapsed time — and elevates the interesting cases: `ERROR` for any response at or above 500, `WARN` for requests slower than two seconds. The threshold is not arbitrary; a request creeping toward the edge timeout is the main server-side warning of a CORS-error window for some browser. Every elevated line carries context parsed from the request: company id, session id, client IP, `CF-Ray`, device id, user agent.

The Anti-Crawler Layer

The receiver sits on the public internet and exists to accept POSTs from any browser on any customer's site, which makes it a magnet for exactly the traffic it does not want: search-engine crawlers that discover the host, and the newer AI-training crawlers that scrape anything reachable. A crawler "session" is worse than wasted bandwidth. It pollutes replay data with fake visits, inflates session counts, and burns storage and processing on nonsense.

The defense has three layers with different audiences. First, `robots.txt` is served at `GET /robots.txt` and disallows everything to every crawler — a wildcard `User-agent: *` group plus explicit per-agent groups covering search engines (Googlebot, Bingbot, YandexBot, and more) and AI crawlers (GPTBot, ClaudeBot, CCBot, PerplexityBot, and more). The duplication is deliberate: some AI crawlers honor only a group that names them explicitly, and the robots standard gives the most specific match precedence over any generic group a CDN prepends. Second, `X-Robots-Tag: noindex, nofollow` is stamped on every response from the host, so anything robots.txt misses is still never indexed. Third — because robots.txt is a request, not a lock — requests whose user agent contains a known AI-crawler fragment are rejected outright with HTTP 403 before reaching any handler.

The nuance is what the layer does *not* block. Search-engine bots are deliberately not hard-blocked: robots.txt fully disallows them, but answering 403 can trigger "soft-404" behavior in their indexers. Unit tests pin the policy down: GPTBot, ClaudeBot, and CCBot user agents get 403; a normal Chrome user agent and even curl pass through; a Googlebot user agent is

let through to be governed by robots.txt. A regression in that table would be a regression in data quality, so the tests exist to prevent one.

Creating and Updating the Session

Middleware done, the handler does session bookkeeping. The `sessions` collection in MongoDB is the coordination point every later service reads and writes, and the receiver is the only service that creates sessions. Each POST is also a chance to refresh the sessions it knows.

When the id is `NEW`, or belongs to a session the platform already marked `FINISHED` (a stale id the recorder kept), the receiver mints a fresh session. It starts `ACTIVE` with a `startTime` of now — that instant is the session's first seen — and records what this first POST can tell: company id, client IP, browser name and OS parsed from the user agent, and the device id. The company document is looked up first, because a session must never be created for a company that does not exist; the lookup accepts the company's object id or its name. If the company carries a preferred storage tenant (`s3ServiceName`), it is copied onto the session at birth: the choice of which archive will eventually hold this session is made now, at ingest, not later by the worker.

The response to a `NEW` request with an empty body is the famous handshake: the receiver writes the fresh session id in the response body, the recorder stores it in `sessionStorage`, and every later batch uses it.

When the id is an existing session, the receiver finds the document, backfills a missing `deviceId` once, and refreshes it — but only up to a point. The refresh obeys a five-second rule: if the session was updated less than five seconds ago, the write is skipped. That throttle is quiet cost engineering: without it, a busy session posting every few hundred milliseconds would drive one MongoDB document update per batch across the whole user base. Instead the session document is written at most once every five seconds, and the update logic decides what the write means: a batch containing any of the six interaction event types (click, navigation, input, scroll, keyboard, mouse move) sets the session `ACTIVE` again and advances its `lastInteractionTime`; a batch of pure background traffic — heartbeats — can nudge it to `IDLE` when a real interaction happened within the last 30 minutes. The receiver is not the arbiter of the full lifecycle — the status job of Chapter 14 owns the timeouts that eventually finish a session — but these updates keep the state machine honest in real time.

The session document is not the only thing that learns about the user. `IDENTIFY` events are read in plaintext *before* payload encryption — the log point carries a small JSON object with optional id, name, and email — and the first one seen stamps the session permanently with that identity, recording that `IDENTIFY` happened so the worker does not repeat the work. For

identified sessions, the receiver keeps a lightweight CRM record warm on the same throttled cadence.

Two things the receiver does *not* do here are easy to assume and wrong. It does not store event payloads in MongoDB; the session document holds metadata only, and the events go straight to the bus. And it does not maintain the session's URL list: the `urls[]` array is assembled downstream, where the worker decrypts `NAVIGATION` events as it archives them. The ingest contract likewise captures no HTTP referrer; the sequence of `NAVIGATION` events is the platform's model of where a user went.

Device, IP, and Location Around Ingest

The browser and OS strings come straight from the user agent, so they are only as trustworthy as that header — not very. A user-agent parser extracts a browser name and an OS name, which are stored, shown in the session list, and used for device filtering; the coarse device class (mobile, tablet, desktop) is derived from the parsed OS on the legacy ingest path.

The IP is stored raw, which makes it the anchor for location. Country and city come from an IP-geolocation provider (ip-api, reached here at `pro.ip-api.com`), cached in MongoDB by IP with a 24-hour time-to-live; TTL and compound indexes keep reads cheap and eviction automatic. Where the lookup runs has shifted during the Go migration — the legacy path resolved per request inside the receiver, the Go rewrite carries the same service and cache — but the durable fact is simple: the receiver captures the client IP, and location derives from that one value anywhere later.

Encrypt, Then Publish

Bookkeeping done, the receiver turns to the payload: each log point's `data` field is encrypted, and the point is published to the bus. The order is fixed — encrypt first, publish second — because after this moment the event leaves the receiver's process and travels through the bus and the archive, where no component should ever see plaintext again. The worker, the processor, and the player API all hold the shared key and decrypt only when they must.

Encryption applies to every log point that carries data, which is most of them: the JSON payloads of clicks, inputs, navigation events, network calls, and so on. Each payload is encrypted with AES under the shared `encryption.key`, base64-encoded, and written back into the log point's `data` bytes before serialization. The rest of the envelope — type, timestamp, order, index — stays plaintext, which is acceptable because the payload bytes are where user content lives. The platform's current mode is one shared symmetric key; per-tenant keys and rotation are named as the obvious future improvement in Chapter 20.

The published unit is one log point, not one batch. For each item in the POST, the receiver marshals the encrypted point back to protobuf, zips those bytes into a tiny single-entry archive, and builds a NATS message with two headers — `cid` for the company, `sid` for the session — addressed to `sessions3-dev.{sessionId}` (the environment prefix follows the deployment: `sessions3-dev` or `sessions3-prod`). Publish is synchronous in that the call waits for the server's acknowledgment that the message is durably stored, but each publish runs in its own goroutine, so the HTTP handler never waits for it. Chapter 10 examines the bus in depth; the receiver's only concern is that the message is on its way.

Then the handler responds: HTTP 200 with the session id in the body, the handshake contract the recorder expects. The response does *not* depend on the bus: if a publish fails — JetStream unreachable, a marshal hiccup — the failure is logged with full context (company, session, event type, timestamp) and the request still returns 200. This is conscious best-effort design, the mirror image of the recorder's retry logic in Chapter 8: a lost batch is tolerable to the recorder, so a lost publish is tolerable to the receiver, and the few events lost in a rare outage are invisible in replay — though never unobserved, because structured logs make every dropped publish countable.

Timeouts: a 503 Before a 524

The receiver runs under two intermediaries that can kill a slow request first: nginx ingress with an origin timeout around 120 seconds, and Cloudflare in front of it with an origin timeout of roughly 100 seconds. When Cloudflare gives up, it answers the browser with its own HTTP 524 page, which carries no `Access-Control-Allow-Origin` header. To the browser, a 524 is therefore not "the server timed out" but a CORS failure: an opaque network error the recorder cannot interpret, let alone retry sensibly.

The receiver refuses to let that happen. The route is wrapped in `http.TimeoutHandler` with a 20-second cap, so the origin answers first: a request that runs past 20 seconds gets HTTP 503 with a plain-text explanation. Because the CORS middleware wraps the router, that 503 carries the CORS headers, the browser sees a clean status, and the recorder can log and retry on its own schedule. The source comments put the priority in order: Cloudflare's error pages carry no CORS headers, the receiver's do, so the receiver must always win the race. A 503 generated by the service is a diagnosis; a 524 generated by the edge is a mystery.

The diagnostics middleware supports the same philosophy from the observability side: a request slower than two seconds produces a WARN line with the full context block, which is how the platform notices a session or company trending toward timeouts long before the cap is hit. Boot markers, request logs, and error lines together mean a report of "the replay stopped recording" can be traced to a window of receiver behavior — or to the absence of any log line, which is itself the finding that the request never reached the origin at all.

What the Receiver Deliberately Does Not Do

The discipline that keeps an ingestion gateway fast is not what it does; it is what it refuses to do. The receiver could, in principle, decode sessions as they arrive, rebuild pages, detect errors, and write finished analytics — and it would be terrible at all of it, because it would do that work for every user of every company in real time, on the critical path of a browser waiting for a response. So it does none of it. It does not reconstruct pages or run replay logic: replay needs the complete ordered stream, and the receiver sees only the latest fragment. It does not run analytics — no error detection, heatmap aggregation, or scroll analysis. It does not touch the cold archive, manage retention, or build derived collections. It does not even decode the log points it forwards; payloads stay opaque ciphertext from encryption until the worker decrypts them. Its entire transformation of event data is encrypt, marshal, zip, publish; everything else is metadata bookkeeping.

That narrowness is the point. Per-request work is bounded — a few MongoDB operations with five-second internal timeouts, one encryption pass per payload, one zip and one bus publish per log point — and everything expensive is pushed to services that can scale and retry independently. The phrase the architecture is built around is a **thin hot path**: the path a live browser touches must contain only the work that cannot be deferred.

Scaling Out

Because the receiver keeps no state of its own, scaling it is close to trivial. There is no session affinity, no in-memory queue, no local cache to warm: every fact a handler needs lives in the request or in MongoDB and NATS, the only shared state in the system. More traffic means more pods behind the ingress, and each new pod boots, connects to Mongo and NATS, logs its pod name, and starts answering POSTs; nothing needs partitioning or rebalancing between instances.

That statelessness makes the platform elastic at the front door. A promo spike that triples a customer’s traffic needs no advance resizing: the autoscaler adds replicas as CPU climbs, and the burst is absorbed in the stateless tier while the durable bus absorbs the matching message burst. The one scaling worry is shared infrastructure — too many replicas writing session updates can load MongoDB, which is exactly why the five-second throttle exists and why payloads go to the bus rather than the database.

Failure Modes

Most requests are healthy, but the failure modes matter because each means something different to the recorder on the other end. A body that does not parse as protobuf — corruption, a wrong content type — produces HTTP 400. A well-formed but unknown

session id produces HTTP 404, as when a stale recorder resumes a session the platform removed. A company that cannot be found fails the create path with HTTP 500 — a session is never minted for a phantom company. Disabled or deactivated accounts are stopped further upstream, at the control plane where billing and company state live, because the ingest endpoint itself must stay open to the public web. Slow MongoDB or a JetStream outage become the logged best-effort behavior above, and a request that outlives 20 seconds becomes the receiver's own CORS-carrying 503.

None of these paths crash the service. The panic recoverer turns the one truly exceptional case — a bug — into a logged 500, and diagnostics make every 500 and every slow request individually visible, searchable by company, session, device, and IP. The receiver is designed to fail softly and loudly: softly for the browser, loudly for the operator.

> **Story checkpoint — Candlewood Books:** On "Shelf Saturday," Candlewood's summer promo triples storefront traffic in one morning. Marta Reyes braces for the recording outages she half-expected; instead the session list keeps filling normally. Behind the scenes the receiver deployment scales out to meet the spike, every batch lands on the bus, and by lunchtime Marta is watching replays of the promo's first confused visitors. She sighs in relief. The platform absorbed the surge at the front door, where surges are supposed to be absorbed.

Chapter takeaways

- The receiver is a stateless ingestion gateway: one POST route, bounded per-request work, and no state beyond MongoDB and the message bus, so it scales horizontally by adding pods.
- The middleware stack — RealIP, CORS-carrying panic recovery, heartbeat, CORS, structured logging, diagnostics, and the anti-crawler layer — runs on every request and guarantees no response ever looks like a CORS failure to a browser.
- Session bookkeeping is throttled: sessions are created on first contact, refreshed at most once every five seconds, nudged `ACTIVE` on real interaction, and never used to store event payloads.
- Event data is encrypted at the receiver before it leaves the process and published to JetStream one log point at a time; the HTTP 200 response is independent of publish success by design.
- The receiver answers with its own CORS-carrying 503 after 20 seconds so Cloudflare's CORS-less 524 error page never reaches the browser, and warns on requests slower than two seconds.
- The receiver deliberately does none of the expensive work — replay, analytics, archive, URL and location enrichment — keeping the live path thin and deferrable.

Chapter 10: The Message Bus: NATS JetStream

Between the receiver that accepts event batches and the worker that archives finished sessions sits a deliberate gap, and that gap is one of the most important pieces of the whole architecture. The receiver runs hot: it answers browsers in milliseconds, at whatever rate the internet happens to be producing traffic. The worker runs cold: it takes its time reading, decrypting, re-chunking, compressing, and uploading, and its throughput is bounded by real work rather than by network arrivals. Connect those two services directly, and the faster one would either stall the slower one or overwhelm it. LogNroll inserts a **message bus** between them — NATS JetStream, running inside the Kubernetes cluster — and the entire pipeline is shaped by the properties that bus provides.

This chapter covers why a session replay platform needs a durable queue between ingestion and archival, how LogNroll chose NATS JetStream over the obvious alternatives, the subject scheme it uses (one subject per session), and how the worker consumes, acknowledges, and finally purges what the receiver publishes. Along the way it becomes clear why the architecture treats the bus as a first-class citizen rather than as plumbing — and what it means to say that **the bus is where sessions wait to be archived.**

Why a Durable Bus at All

The first reason is decoupling: the receiver and the worker must not care about each other's speed. Recording traffic is bursty — a normal Tuesday trickles, a promo or a launch spikes many times larger, without warning — and the receiver is exactly where the platform does not want to do heavy work or drop data. If it handed each batch directly to an archiver it would inherit the archiver's bottlenecks: slow uploads would slow ingestion, a worker restart would block live recording, and the one component that must stay thin and fast would spend its time waiting.

Durability is the second reason, and it separates a bus from a plain in-memory queue. Messages are written to disk by JetStream before the receiver's publish call is acknowledged. A worker that crashes mid-archival, a pod that is evicted, a deploy that restarts the fleet: none of these lose data, because the messages were never in a worker's memory — they sit in the stream until the next healthy worker picks them up. That matters doubly for replay: the events in a session are irreplaceable, so the pipeline between browser and archive should be the most durable part of the system, not the least.

Replayability follows from the first two. Because the bus keeps messages after a consumer has seen them, work can be retried and re-read: a session that fails to archive can be claimed again later and drained again, and a worker bug can be fixed with the backlog replayed — no

user ever has to revisit a page. None of this is possible with a fire-and-forget pipe, and none of it requires the receiver to store anything beyond the moment of publish.

There is also an operational reason. The bus makes the ingestion-to-archival boundary *visible*: backlog is measurable, and a growing backlog is an early, legible warning that the worker tier needs capacity — far easier to see than a subtle slowdown inside a service.

Choosing the Bus: NATS vs Kafka vs Redis

NATS JetStream is one of three widely used answers to the durable-queue question, and it is worth a brief, fair comparison because the choice shapes operations for years. The three families differ mainly in how much machinery they bring and what they optimize for.

Apache Kafka is the heavyweight: a distributed, partitioned commit log, with messages replicated across brokers and consumed under strong per-partition ordering and explicit consumer groups. It shines at very high throughput, long retention, and many independent consumers of the same stream, and it is the default when an organization already runs a data platform around it. The cost is operational — a cluster to run, tune, and feed — and for a message rate measured in individual log points it is more machinery than the job requires.

Redis Streams are the lightweight end. `XADD` and `XREADGROUP` give a pub/sub-style list with consumer groups and pending-entry tracking, and Redis is fast, familiar, and easy to run. The caveats are durability and scale: data lives in memory, persistence is a configuration choice rather than the core guarantee, and a busy stream competes with everything else the instance does.

NATS JetStream sits in the middle, and it is what LogNroll runs. NATS itself is a small, fast publish/subscribe server; JetStream layers persistence on top — streams, consumers with acknowledgments, retention policies — without introducing a second system to operate. For traffic measured in individual log points, JetStream offers the durability of a disk-backed log with the operational profile of a single modest cluster, which is exactly the trade the architecture wants. The choice is not universal: a much larger platform with a streaming-data culture might reasonably live on Kafka, and a very small one that tolerates losing the last seconds of a session might use Redis Streams. LogNroll needs data to survive worker failures, so the durable middle ground is home.

One Subject per Session

JetStream organizes messages by **subject**, a dot-separated name, and stores subjects in a **stream** — a named, durable message log that can match a subject pattern. LogNroll's scheme is simple and slightly unusual: the subject *is* the session. The receiver publishes to `sessions3-dev.{sessionId}` (or `sessions3-prod.{sessionId}` in production; the environment

prefix keeps development traffic completely out of the production stream), and all of those subjects are captured by a single stream named `sessions3-dev-stream`, configured to match `sessions3-dev.*`, stored on disk, and created automatically on boot if it does not exist yet.

```
receiver (N replicas)
   │  publish per log point, subject sessions3-dev.<sessionId>
   │  message = zipped LogPoint, headers: cid=<companyId> sid=<sessionId>
   ▼
JetStream stream  sessions3-dev-stream   (file storage, limits retention)
   ├── subject  sessions3-dev.64b7…a1   ← one session's whole event history
   ├── subject  sessions3-dev.64b7…b2
   ├── subject  sessions3-dev.64b7…c3
   └── ... one subject per session, in publish order
   ▼
worker (pull consumers)
   │  claims FINISHED session → PullSubscribe its exact subject
   │  Fetch in batches → unzip → stage to disk → Ack (or Nak on error)
   └─ after archiving: purge that subject, session gone from the bus
```

Each message is one log point, not one batch: the receiver loops over the items in a POST and publishes them individually, so a two-hundred-event batch becomes up to two hundred small messages. Every message is a zipped, serialized `LogPoint` — a single `logpoint.pb` entry wrapping the encrypted payload from Chapter 9 — carrying two headers: `cid` for the company, `sid` for the session. Subject and headers are deliberately redundant: the subject routes, and the headers let the worker log and file each message without re-deriving context.

Why one subject per session instead of shared subjects? Because the session is the unit of ordering, consumption, and cleanup. Ordering: every message for a session lands in the same subject in the order the server accepted it, so draining it yields the session's events as one contiguous run, with no partitioning or sort step at consume time. Replay correctness does not lean on this alone — publishes are concurrent, so acceptance order can interleave slightly, and each `LogPoint` carries its own timestamp and sequence for the player to sort on — but subject ordering is a convenient default. Consumption: the worker archives one session at a time, so it subscribes to exactly one subject, drains it, and moves on; sessions never contend on a shared queue. Cleanup: archiving ends with purging that subject, a surgical removal of one session's messages rather than a filter across a shared stream.

Streams, Retention, and the Disk

The stream is where durability lives. When the receiver calls JetStream's publish, the server appends the message to the stream on disk and only then acknowledges it, which is what makes the receiver's fire-and-forget goroutine safe: the acknowledgment it waits for means "stored," not merely "received." A JetStream stream is not an in-memory buffer that evaporates when the cluster restarts; it is a file-backed log with its own retention policy, and the messages it holds survive consumer crashes, worker restarts, and redeploys.

The retention policy is the default **limits policy**: messages are kept until they hit configured limits or are explicitly removed. Under limits retention, a message is *not* deleted when a consumer acknowledges it — acknowledgment tells the stream that this consumer is done, so the message is not redelivered to it, but the message itself remains until limits or an explicit purge remove it. The stream is a log, and logs accumulate. That is why the worker finishes each session with an explicit purge, described below; without it, every acked event would sit in the stream forever and the bus would grow without bound. Operators can also inspect streams and purge them wholesale with the `nats` command-line client when something goes wrong — exactly the control an operator wants over a durable log.

Pull Consumers and Acknowledgments

JetStream offers two consumption models. Push consumers receive messages as the server delivers them; **pull consumers** — the model the worker uses — ask for messages when they are ready, in batches they choose. For an archiver the distinction matters: a worker busy uploading a large session to object storage should not be force-fed messages mid-upload. Pulling puts the pace in the worker's hands.

The worker's consumption loop is built around a single session at a time. Once it has claimed a `FINISHED` session (the locking dance belongs to Chapter 11), it opens a pull subscription on that session's exact subject and fetches messages in batches — by default up to 20 at a time, waiting at most 10 seconds for the batch to fill — until the subject is drained. Each message is unzipped, decoded, and written to local staging before it is acknowledged, and the ordering is deliberate: **acknowledge only after the durable local write**. Crash between write and ack? The message is redelivered later. Ack before write? A crash could lose an event the bus had already forgotten. A message that cannot be unzipped or decoded is negatively acknowledged (`Nak`) so JetStream redelivers it rather than silently dropping a salvageable event. When the subject is empty, the worker unsubscribes; the consumer was ephemeral, created for this session and discarded with it, so no consumer state accumulates across sessions.

At-Least-Once Delivery, on Purpose

LogNroll deliberately runs on **at-least-once** semantics: in normal operation every message is delivered and archived exactly once, but under failure a message can arrive more than once, and duplicates are designed to be harmless. Exactly-once is achievable only at real cost — deduplication keys on every message, transactional coordination between bus and archive, idempotency machinery at both ends — and replay payloads are the worst candidate for that expense: a duplicated mouse-move event, or even a duplicated archive chunk, is invisible to the person watching the replay.

The scenario the design rehearses for is a worker dying mid-archival: it has drained half a session's subject, acknowledged those messages, uploaded nothing, and crashes. The session's Mongo lock expires after about two minutes, another worker claims the session, opens a fresh consumer on the same subject, and — because acknowledged messages are still in the stream under limits retention — reads the history again. Some events are processed twice; nothing breaks, because the downstream steps are idempotent by construction: re-reading rebuilds the same staging files, archive chunks are named deterministically from the same timestamp windows so re-uploads overwrite identical objects, and the `processedTimestamp` checkpoint advances only as far as the work actually completed. At-least-once plus idempotent consumers buys crash safety without the coordination tax of exactly-once — the right trade for data whose value is completeness, not transaction boundaries.

Purging a Subject After Archiving

When archival of a session completes — every message drained, every chunk uploaded to object storage — the worker performs one final, load-bearing act: it purges the session's subject from the stream. In JetStream terms this is a stream purge filtered to one subject: delete every message under `sessions3-dev.{sessionId}`, and nothing else. The session's messages have served their purpose; they now exist, more permanently and more cheaply, in the cold archive, and keeping them on the bus would serve no one.

The purge exists because of the retention semantics above: limits retention does not delete acknowledged messages, so the only way the bus stays bounded is explicit, per-session removal. Without the purge, every event from every session would accumulate on the bus — a redundant second copy of the archive on JetStream's disk, growing without limit and turning the stream into an accidental retention system with no policy. The purge keeps the stream's population equal to the sessions *in flight*: published but not yet archived. That is the correct steady state for a waiting room, and it is why the bus never needs the capacity planning a permanent log demands. (The scheme's predecessor — subjects like `log_points` and `log_points_batch` without JetStream persistence — survives only as disabled configuration, a reminder that the current design is the platform's second, deliberate attempt at this boundary.)

Backpressure and Sizing

With durable storage, pull consumers, and per-session purge in place, the bus behaves as the pipeline's shock absorber. Receivers publish as fast as browsers arrive, and the stream absorbs the burst; workers drain at the pace they can sustain, and a burst simply grows the backlog for a while. The bus is where sessions wait to be archived, and a waiting room is

exactly the right metaphor: it exists precisely so that a flood of arrivals does not force the people working the counter to work faster than they safely can.

Sizing follows from that role. Stream capacity is disk and I/O rather than memory, so a promo spike shows up as growing stream storage and message backlog — both measurable — rather than as dropped messages. The worker tier is sized against the backlog: more sessions waiting means more replicas, each draining a session at a time. The receiver never blocks on the worker; its publish acknowledgments are paced by the stream's ability to store, and if the stream is truly unavailable, publishes fail loudly and are logged, leaving the recorder's own retry logic (Chapter 8) as the last line of defense. The environment prefix completes the picture: dev traffic accumulates in `sessions3-dev-stream` and production in `sessions3-prod-stream`, so a runaway dev recorder can never crowd out real sessions.

None of this is exotic. It is a durable log, one subject per logical unit of work, pull-based consumption, at-least-once delivery with idempotent downstream handling, and disciplined cleanup. But assembled in this order, those five choices are what let the platform promise that a session, once accepted, will be archived — eventually, correctly, and without anyone having to choose between a fast receiver and a careful worker.

Chapter takeaways

- A durable bus between receiver and worker decouples bursty ingestion from archival, survives worker restarts without data loss, and makes the ingestion-to-archival handoff visible and retryable.
- NATS JetStream is the middle path: disk-backed durability like Kafka without the cluster machinery, and stronger persistence than Redis Streams; the choice fits a workload of one message per log point.
- The subject scheme is one subject per session — `sessions3-{env}.{sessionId}` on stream `sessions3-{env}-stream` — making per-session ordering, consumption, and cleanup trivial.
- The worker uses pull consumers, fetching batches of messages and acknowledging each only after it is durably staged, so a crash redelivers rather than loses events.
- Delivery is at-least-once by design; duplicate processing is harmless because claiming, staging, chunk naming, and checkpoints are idempotent downstream.
- Subjects are purged after archiving because limits retention keeps acknowledged messages; without the purge the stream would grow into an unbounded, policy-free second archive.

Chapter 11: The Worker and the Cold Archive

By the end of the last chapter, a finished session is sitting on a message bus: every log point the browser sent has been accepted by the receiver, encrypted, zipped, and parked on a JetStream subject named after the session id. That is a comfortable place to wait. It is not a comfortable place to live forever. JetStream storage is fast and durable, but it is neither free nor infinite, and every message sitting there is holding a CPU core, some memory, and a slice of NATS disk in a session replay service that would rather spend those resources on sessions that are still being recorded.

This chapter is about the worker — the service that drains that queue. Its job is to take a finished session out of the message bus and file it into the **cold archive**: the object store where a session's log points will live for the rest of their retention period, cheaply and at rest. In LogNroll this is a single Go service, `lognroll-worker-go`, whose heart is one file, `archiver.go`. Everything the worker does — and everything it deliberately does not do — follows from a simple division of labor. The receiver is built for throughput on the hot path: it must accept the next batch of events from a browser within milliseconds and never slow the checkout page it is watching. The worker is built for the opposite job: heavy, multi-step, failure-prone work that nobody is waiting on. Decoding again, re-chunking, compressing, uploading, deleting — none of that belongs on the hot path. It belongs in a background service that can retry, be restarted, and be scaled horizontally without anyone noticing.

The worker is also the quiet custodian of the platform's promise. Once a session is archived, the processor can analyze it, the player API can serve it, and the dashboard can list it — but none of that can happen until the worker has done its job. In the story of this book, the worker is the librarian who files every finished session onto the shelf. This chapter follows one session through that worker's hands, step by step.

The claim dance

A worker that processes sessions needs work, and the work is defined by a deceptively small query. Every sweep, the worker asks MongoDB for a session that matches three conditions, all on the shared `sessions` collection: the status must be `FINISHED`; the `storageName` field must not be `"SPACES"`; and the session must not currently be locked — or its lock must be stale.

The first condition is the lifecycle handshake from the previous chapters: only a session the status job has declared finished is ready to archive. The second is worth pausing on, because `storageName` is doing double duty. When the receiver creates a session, no storage location has been chosen yet and the field is empty. LogNroll treats that as meaning "this session's log

points still live on the message bus." When the worker finishes archiving, it sets `storageName = "SPACES"`, meaning "this session's log points now live in the S3-compatible object store." So the very same field that records *where* a session's content lives also acts as a pipeline marker: `storageName != "SPACES"` reads, in effect, as "not yet archived." The player API and the session processor rely on the same trick in reverse — they only touch sessions whose `storageName` is `"SPACES"`, because only those have content in the object store. One string, three services, one meaning: has this session reached the archive?

The third condition is the **lock pattern**, the mechanism that keeps the whole archival pipeline honest. Archiving is not atomic: it reads, transforms, and uploads over minutes, and more than one worker process may be running at once. If two workers grabbed the same session, they would both read the same messages and upload duplicate archives. So claiming a session is a single atomic operation, a MongoDB `findOneAndUpdate` that both matches the session *and* stamps it as taken, in one round trip:

```
// archiver.go — the claim filter and the atomic lock (condensed)
func (w *WorkerArchiver) getFilter() bson.M {
    twoMinutesAgo := time.Now().Add(-2 * time.Minute)
    return bson.M{
        "status":      "FINISHED",
        "storageName": bson.M{"$ne": "SPACES"},
        "$or": []bson.M{
            {"lockedBy": nil},
            {"lockedBy": bson.M{"$exists": false}},
            {"lockTimestamp": bson.M{"$lt": twoMinutesAgo.Format(time.RFC3339)}},
        },
    }
}

func (w *WorkerArchiver) LockPendingSessionJob(ctx context.Context) (*models.UserSessionEn
tity, error) {
    update := bson.M{"$set": bson.M{
        "lockedBy":      w.instanceID,                 // unique per worker process
        "lockTimestamp": time.Now().Format(time.RFC3339),
    }}
    var session models.UserSessionEntity
    err := w.db.Collection("sessions").
        FindOneAndUpdate(ctx, w.getFilter(), update).Decode(&session)
    if err == mongo.ErrNoDocuments {
        return nil, nil // nothing to do right now
    }
    return &session, err
}
```

The update writes two fields: `lockedBy`, set to a unique id the worker generated for itself when it started (a fresh MongoDB `ObjectId` hex string per process), and `lockTimestamp`, the moment the claim was made. The filter's `$or` clause is what makes the lock safe against crashes. A session is claimable if it has never been locked, or if its lock is older than the **lock expiry**, which the worker sets at about two minutes. If a worker dies mid-archive, it cannot run its own cleanup, so its claim would otherwise sit on the session forever; with the two-minute expiry, the session simply becomes claimable again by the next sweep, and a healthy

worker picks it up and finishes the job. Lock expiry is the difference between “distributed processing” and “distributed deadlock”: it is the system agreeing, in advance, that any claim older than two minutes is abandoned and up for grabs. A separate value lives on the session record too — `retries` — which counts how many times archiving has been attempted; the worker treats more than ten attempts as a sign that the session is broken, not merely unlucky.

> **Field note:** Think of the worker as a librarian. The lock pattern is “claim the book from the return cart, file it on the shelf, put the claim slip back.” If a librarian is interrupted halfway, the slip goes stale, and after two minutes anyone else may pick the book up and finish shelving it.

From subject to staging files

Once the worker holds a claim, the real work begins. The session’s log points are still on the bus as individual JetStream messages on the subject `sessions3-{env}.{sessionId}`, and they need to become files on the worker’s local disk first. That staging step is `ReadAllSubject`, and it is deliberately modest: the worker opens a pull subscription on the session’s subject, fetches messages in batches (up to twenty at a time, waiting at most ten seconds for more), and for each message does three things — unzip it, unmarshal it, and write it to disk.

Recall from the receiver chapter what a bus message actually contains: one log point, serialized as protobuf, wrapped in a zip archive with a single entry. The unzip step is therefore cheap and mechanical. What comes out is a `LogPoint` — one event from the session, with its timestamp, type, version, and order fields intact, and its payload `data` still encrypted from the receiver. The worker does not decrypt anything here; it simply writes each log point to its own file on local disk, named `{timestamp}-{uuid}.dat`, under a folder per session: the operating system temp directory, then the company id, then the session id.

Why write one file per event instead of accumulating the events in memory? Two reasons. First, memory discipline: a long session can hold tens or hundreds of thousands of events, and a worker processes sessions one after another all day. Streaming events to disk keeps the working set small no matter how large a session turns out to be. Second, durability of progress: the worker acknowledges each message to JetStream only after its file is safely written, and it purges the subject only at the very end of the whole archive operation. If the worker crashes in the middle, the acknowledged messages are gone from the bus but the files are on disk, and the retry machinery from the claim dance can resume from where the session’s `processedTimestamp` checkpoint says it got to. Disk, in other words, is the worker’s private staging area: big enough to hold one session at a time, wiped as soon as the upload succeeds.

Re-chunking: turning a stream into frames

Now the worker holds the session as a sorted pile of per-event files on disk, each named with its event timestamp. The next step is the worker's most interesting decision, because it changes the shape of the data permanently: it **re-chunks** the event stream into fixed-size frames.

This is worth stopping on, because the receiver already batched events once — batches of up to two hundred events, sent every hundred milliseconds or so. Why would the worker not just forward those batches to the object store? Because the receiver's batches are shaped for the network, not for the archive. A network batch is whatever happened to be queued when the timer fired: a few mouse moves here, a burst of mutations there, with no meaningful relationship to time or size. An archive object, by contrast, wants two properties: it should correspond to a clean slice of the session's timeline, so that a reader can say "give me the events around the checkout click," and it should be a sane size, so that no single object is enormous and no single object is so small that a session fragments into thousands of wasteful files.

The worker's frame builder walks the files in timestamp order and groups them into windows of about sixty seconds, growing a window as needed until it holds a meaningful amount of data — the code's `fileMaxSize` is five million bytes. Each frame is then marshaled into a protobuf `LogPoints` batch and compressed into a fresh zip archive. The inner zip entry is named `{frameEnd}.logpoints.pb`, where `frameEnd` is the end timestamp of the frame. The result is handed to the object store through a single method, `UploadSessionLogPoints(sessionID, filename, zippedPoints)`, which stores the bytes under a key that groups every frame of a session together.

```
sessions/{sessionId}/
    {frameEnd}.0.logpoints.json
    {frameEnd}.1.logpoints.json
    {frameEnd2}.0.logpoints.json
    ...
```

The `.json` suffix is a small piece of archaeology: it dates from when the archive held zipped JSON and the name was kept for compatibility after the format moved to protobuf — the objects are zips of binary protobuf today, not JSON. What matters is the timestamp prefix. Because every object key begins with its frame's end timestamp, listing the folder and sorting the keys puts the session's frames in chronological order for free, and a reader can fetch exactly the frames it needs.

During the walk, the worker also does a little bookkeeping that the receiver used to do and now leaves to the archive pass. For `IDENTIFY` and `NAVIGATION` log points it decrypts the payload just long enough to record the user's identity or the page URL on the session document — one URL per navigation, identity stored once per session. For `STYLES` events it

goes further: if the recorded style payload is a URL, it fetches the stylesheet, encrypts the full CSS, and stores it back in the payload, because the player needs the actual styles to reconstruct the page. Every other event's payload passes through still encrypted; the player will decrypt it when the session is watched, which is the last line of defense for the recorded data at rest.

Two tenants, one interface

The worker does not know — or care — which physical bucket it is writing to. Its S3 access is hidden behind a tiny interface with one method, `UploadSessionLogPoints`, and at startup it builds a map of implementations from configuration, one per storage tenant. The choice is recorded on the session itself: the receiver copies an `s3ServiceName` from the company record when it creates the session, and the worker simply looks that name up in its map. LogNroll runs two tenants today, mirroring its two customer groups: the default tenant writes to the `lognroll-test` bucket in DigitalOcean Spaces' Frankfurt region (`fra1`), and the second tenant, `s3SpacesServicePOLISUA`, writes to the `sessionreplay` bucket in Amsterdam (`ams3`). Tenants exist so that one customer group's storage, billing, and access keys never mingle with another's — a quiet piece of multi-tenancy that costs almost nothing at this layer, because the worker only ever sees a string and a map.

Upload, purge, unlock

The frame loop runs until every staged file has been folded into a frame and uploaded. Then the worker performs the finishing sequence, and the order of those steps is a small lesson in at-least-once delivery. First it removes the session's local staging folder — the session's private scratch space has served its purpose. Next it purges the session's subject from JetStream, deleting every remaining message for that session id in one call. Only after the archive is confirmed on the object store does the worker delete the bus copy; if the upload had failed, the messages would still be there, untouched, waiting for the retry. Finally, the worker releases its claim and flips the session's `storageName` to `"SPACES"` — the migration that tells the processor, the player API, and the dashboard that the session's content now lives in the archive. From this moment the session is archived: readable from the object store, no longer occupying the bus.

One more real detail completes the picture. For sessions where the user was identified, the worker also updates LogNroll's CRM-style users directory: it claims a one-time counter on the session atomically so that exactly one component — the worker now, or the app API's live sync later — ever increments the user's session count and total duration, and then it writes the user's latest session summary. It is a small, self-contained piece of bookkeeping, but it is

a good example of the pattern running through this entire architecture: whenever two components might do the same accounting, one of them must win the claim first.

Failure paths: retries, FAILED, and the honest archive

Archiving is real work with real failure modes, and the worker's error handling is worth reading because it shows what the platform considers recoverable. A failure to read the bus, or a session folder that turns up empty, calls `FailSession`, which increments the session's `retries` counter and leaves the claim in place. The claim will expire in about two minutes, and the next sweep will pick the session up again and try once more — a slow, self-healing retry loop with no timers of its own. A failure during an upload simply aborts the pass the same way; because the bus subject is purged only after success, the messages are still there for the retry.

Two conditions are treated as permanent, and both funnel into a path the code calls "failed and removed": if a session has been retried more than ten times, or if its record is missing the last update time entirely (a sign the session never really got going), the worker stops trying. It purges the subject, releases the lock, deletes the local staging folder, marks the session `FAILED`, and counts it in its failure metrics. A `FAILED` session is the platform being honest: not every session that is recorded can be archived, and it is better to say so explicitly in the data than to leave a session retrying forever. One operational comfort: because the worker writes its progress back to the session as a `processedTimestamp` after every frame, even the recoverable path is rarely a full restart — a retried session resumes after the last frame that was actually uploaded, rather than re-uploading everything from the beginning.

```
ACTIVE     ──►  IDLE        batches keep arriving, no user interaction (receiver)
ACTIVE     ──►  FINISHED    timeouts hit (status job)
IDLE       ──►  FINISHED    timeouts hit (status job)
FINISHED   ──►  archived    worker claims + archives; storageName = "SPACES"
FINISHED   ──►  FAILED      too many retries
archived   ──►  REMOVED     retention elapsed (remover job)
```

Why chunked zips, again

The value of the frame layout becomes clearest when you look at who reads the archive next. The session processor wants to analyze the whole session, so it lists the session's frames in key order and walks the events. The player API does the same when a replay is requested: because every object key begins with its frame's end timestamp, sorting the listing yields the session's events in order, and each object is small enough to fetch and decompress without one enormous transfer. Any reader can also fetch a single frame by its key — say, only the minutes around a bug report — without touching the rest of the session. That property is what keeps the door open to the lazy, per-frame fetching the player chapters discuss, and it is

exactly why the frames carry timestamps in their names rather than opaque sequence numbers. Chunked zips also keep the object store healthy: no single object grows without bound, uploads stay small enough to be retried cheaply, and a session that would have been one fragile, hour-long archive is instead a handful of independent, replaceable parts. If one frame is corrupted, one frame is re-archived — the rest of the session is untouched.

The worker as a service: metrics, shutdown, and scale

Archiving is a background service, but it is still a service, with the operational trappings to match. LogNroll's worker exposes a health endpoint and Prometheus metrics on its actuator port: a gauge of how many sessions currently match the archive filter (`lognroll_sessions_count`), counters for sessions processed and failed, and a histogram of session processing time. Those metrics are not decoration. In the production deployment, the worker runs as a small autoscaled deployment whose horizontal pod autoscaler watches that gauge: when finished sessions pile up faster than the workers can archive them, the count rises and Kubernetes schedules more worker pods; when the backlog drains, they scale back down. The claim dance exists precisely so that this scaling is safe — any number of worker pods can run the same sweep, and the atomic lock guarantees that each session is archived exactly once, no matter which pod wins the claim.

Graceful shutdown gets the same care as scaling, because a worker that is killed mid-archive must not lose work. On `SIGTERM` the process flips a stopping flag and refuses to claim new sessions; the archive pass checks that flag at its safe points, and if it is set, the worker cleans up its staging folder and releases its claim so that another pod can finish the session. The main loop then waits for the in-flight sweep to drain — up to two minutes if a long session is still being processed — before the process exits. Inside that window, the pod's autoscaler and the Kubernetes deployment controller have already noticed that the pod is going away and are standing up a replacement — which will pick up any released claim two minutes later, none the wiser.

Look back at what happened to one session in this chapter. It arrived as a stream of individual bus messages; it left as a handful of timestamped, compressed frames in an object store, with its metadata updated, its bus copy deleted, and its record pointing at the archive. It went from being a live, expensive thing to being a filed, cheap thing — and nothing downstream had to change its behavior to make that happen. That is the worker's quiet triumph: the cold archive is cold not because the data is unimportant, but because the platform has already done all the expensive work of making it easy to find again.

Chapter takeaways

- Archiving is deliberately off the hot path: the receiver never decodes, re-chunks, or uploads; it parks encrypted events on the bus and lets the worker do the heavy lifting asynchronously.
- Claims are atomic and expiring: a `findOneAndUpdate` lock with a roughly two-minute expiry lets any number of worker pods race safely, and a crashed worker's claim simply becomes stale and reusable.
- `storageName` doubles as a pipeline marker: empty or non-`"SPACES"` means "still on the bus, not yet archived," and `"SPACES"` tells the processor and player API that content is in the object store.
- The worker re-chunks the event stream into time-bucketed frames of roughly sixty seconds and up to about five megabytes, so every archive object is bounded, chronologically named, and independently replaceable.
- The bus subject is purged only after the upload succeeds, and progress is checkpointed per frame, so retries resume where they left off and a crash never loses an archived session.
- Metrics are load-bearing: the count of sessions awaiting archive drives autoscaling, and a graceful shutdown releases in-flight claims so replacements can finish the work.

Chapter 12: Storage, Lifecycle, and the Cost of Memory

Every session replay platform is, underneath the product, a storage problem with a story attached. A session arrives as an event stream, and the platform must decide, moment by moment, where each piece of that stream belongs: which store is cheap enough to hold bulk data for weeks, which store is fast enough to update on every batch, and which store will answer the dashboard's questions when a support agent searches for a session. The answer is almost never "one store." The previous chapter showed how a session ends up in the cold archive; this chapter zooms out and looks at the whole storage picture — the two stores that divide the work between them, the document that coordinates it all, the lifecycle that moves data from hot to cold to gone, and the cost math that quietly decides how long "gone" waits.

It is easy to imagine a session replay platform as one giant database. It is more accurate to picture a library: a card catalog and the stacks are different systems with different jobs, and the catalog card is not the book. LogNroll's version of that split is between **MongoDB** and **S3-compatible object storage**, and the discipline of the whole architecture is keeping the two roles separate.

The coordination store and the content store

Think about what a session actually needs from storage at each stage of its life. While recording, the platform must update the session's metadata constantly: a new batch arrived, the user navigated to another page, the last interaction moved forward, the device was this browser on that operating system. Those updates are tiny, they happen on the hot path of every request, and they must be consistent — two receivers racing to update the same session must not lose each other's writes. That is the profile of a document database, and it is what the `sessions` collection in MongoDB is for. LogNroll treats it as the platform's **coordination store**: one shared collection, a few kilobytes per document, updated thousands of times a minute across the platform.

The log points themselves are a different kind of data. They are numerous, bulky, binary, and compressed; nothing in the product needs to query them transactionally; and they must survive for weeks at minimal cost. That is the profile of the **content store**. S3-compatible storage is cheap per gigabyte, built to hold arbitrarily large blobs, and organizes data by key prefix rather than by query. LogNroll keeps the archive there, one folder per session under the `sessions/{sessionId}/` prefix, and no service ever stores a log point inside MongoDB.

Both sides of the split are tuned for their role. The coordination store is written defensively: the receiver does not update a session document on every batch — it skips the write if the last

one happened less than five seconds ago — so a session that is generating hundreds of batches per minute still costs the database a handful of small updates per minute, not hundreds. The collection carries a plain index on `companyId`, the field every dashboard query filters by, and that is nearly all the indexing the hot path needs. The content store, for its part, never sees a partial session: it only ever receives complete, frame-shaped archives from the worker, so its access pattern is write-once, read-many, delete-after-retention — the workload object storage vendors designed for.

Why not put everything in MongoDB? Three reasons, and they compound. First, size: MongoDB caps a single document at sixteen megabytes by default, and a busy session's archive is routinely many times that — the data would have to be sliced into fragments anyway, which is exactly what the object store's folder-per-session layout does natively. Second, cost and write amplification: MongoDB in production runs on fast storage with a price per gigabyte an order of magnitude above object storage, and a document holding the whole event stream would be rewritten, in full, every time a batch arrived — the platform's write pattern would be dominated by copying megabytes to add kilobytes. Third, access pattern: the processor and the player read whole sessions, often sequentially, which is precisely what object storage is built for. Keep small, hot, queried documents in MongoDB; keep big, cold, read-in- bulk blobs in the object store. The architecture is healthier for never confusing the two.

The session document

The seam between the two stores is the session document itself. It lives in the shared `sessions` collection, and every service that touches a session — receiver, worker, processor, the jobs, the player API, the dashboard — reads and writes the same collection. The document is a card-catalog entry: enough to list, filter, authorize, and locate a session, with pointers to where its content lives, never the content itself. Its fields follow the session's life closely:

```
{
  "_id": "665f0a1b2c3d4e5f6a7b8c9d",
  "companyId": "64e0f1a2b3c4d5e6f7a8b9c0",
  "deviceId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "startTime": "2026-05-02T14:03:11.200Z",
  "endTime": "2026-05-02T14:21:47.903Z",
  "urls": ["https://shop.candlewood.example/", "https://shop.candlewood.example/
checkout"],
  "browser": "Chrome",
  "os": "macOS",
  "ip": "203.0.113.24",
  "country": "DE",
  "city": "Berlin",
  "status": "FINISHED",
  "storageName": "SPACES",
  "s3ServiceName": "s3SpacesService",
```

```
  "lastInteractionTime": "2026-05-02T14:21:40.110Z",
  "lastUpdateTime": "2026-05-02T14:21:47.903Z",
  "duration": 1116643,
  "events": ["NAVIGATION", "MUTATION", "CLICK", "IDENTIFY"],
  "numberOfInteractions": 214,
  "processedTimestamp": 1752438231000,
  "retries": 0,
  "lockedBy": "665f0f0f0f0f0f0f0f0f0f0f",
  "lockTimestamp": "2026-05-02T14:22:02.000Z"
}
```

Every field earns its place. `companyId` is the multi-tenant boundary — every product query is scoped to it, and the player API authorizes reads by checking the caller's membership in the session's company. `urls`, `deviceId`, `browser`, `os`, `ip`, `country`, and `city` power the dashboard's filters and the device-chain view that connects one visitor's sessions. `duration` and `numberOfInteractions` summarize the session for lists. `events` is a running set of the event kinds seen so far, used to answer "does this session have any clicks?" without touching the archive. `processedTimestamp` and `retries` are the worker's checkpoint and scoreboard from the previous chapter. And two fields — `storageName` and `s3ServiceName` — record *where the content is*: which storage tenant holds the archive, and whether the archive exists at all. The session document is not the session; it is the session's address.

That division explains a subtle fact about the product: almost everything a dashboard shows comes from the card, not the book. Filtering sessions by user, device, URL, or status is a MongoDB query over small documents; counting them, paging them, and rendering their summaries never touches the object store. Only when someone actually opens a session to watch it does the pipeline reach for the archive — and even then it fetches frames, not the whole visit. This is why the metadata document is worth keeping rich: the coordination store is what makes a replay platform searchable, and the content store is what makes it affordable.

The lifecycle: who moves the session, and when

The status field is the session's position in a small state machine that the whole platform agrees on, and it is worth being precise about who performs each transition, because the answer is spread across four different services:

```
receiver        ACTIVE    session created; refreshed while user interactions keep arriving
receiver        IDLE      updates keep arriving but no interaction recently
status job      FINISHED  no updates for a while, or idle too long, or the session ceiling
hit
worker          archived  log points moved to object storage (storageName = "SPACES")
worker          FAILED    archiving failed too many times
remover job     REMOVED   FINISHED session older than the retention cutoff; content
deleted
```

A session is born `ACTIVE`. The receiver creates it on the first POST, stamps `startTime`, and revisits the document on every subsequent batch: it refreshes `lastUpdateTime`, and if the batch contains interaction events — clicks, navigation, input, scroll, keyboard, mouse movement — it sets the status to `ACTIVE` and refreshes `lastInteractionTime`. A batch without interactions can soften the status to `IDLE`: the visitor is still on the page, still connected, but not doing anything. This is the receiver's half of the lifecycle, and it is deliberately simple — it only ever nudges a session toward the present.

Deciding that a session is *over* is a separate job, because "over" is a judgment call, not an event. A visitor who closes the tab leaves no goodbye packet; the platform must infer that the visit ended from silence. That inference belongs to the **session status job**, a small Go program scheduled as a Kubernetes CronJob every minute. Each run loads the sessions still in `ACTIVE` or `IDLE` and applies the same three rules: a session is finished when no update has arrived for two minutes — silence that long means the browser stopped talking; when no interaction has happened for thirty minutes — the tab may be open, but the human is gone; or when the session has run past a hard ceiling of three hours since `startTime` — no visit is allowed to linger forever. Whichever threshold trips first stamps the session `FINISHED`, and the job sets `endTime` and `duration` from the timestamps it has. These constants are real code values, and they are tuning knobs, not laws; what matters is the shape — a background job, running every minute, turning silence into a status change.

`FINISHED` is the handoff point. The worker from the previous chapter claims finished sessions and archives them, which flips `storageName` to `"SPACES"` and empties the message bus. The session has now moved from the coordination store's responsibility to the content store's. From here, nothing happens to it until one of two endings: the processor reads it and the player serves it for as long as it lives, or the **session remover job** deletes it when its retention window elapses.

Retention as a product decision

The remover job is where storage policy becomes visible, and it is worth understanding exactly what it does and does not do. On its schedule it asks MongoDB for a `FINISHED` session whose `startTime` is older than the retention cutoff — the code computes "a month" as thirty-one days, a little slack past the round thirty that the product advertises. For each match it deletes every object under the session's `sessions/{sessionId}/` prefix from the object store, sets the status to `REMOVED`, and moves on. The metadata document survives: a `REMOVED` session still has its card in the catalog — the row that keeps ids stable and answers "no, this session is gone, and here is why" — but its content is irrevocably deleted. From the product's point of view, a removed session is gone.

The striking thing about that job is how much product policy hides inside an engineering constant. Thirty days is not a number that fell out of a database manual; it is a promise made to customers and, through them, to the people whose sessions are recorded. GDPR-style thinking pushes retention down — data minimization means not keeping recordings longer than you need them. Incident response pushes it up — the checkout bug found in November is useless if September's sessions are gone. LogNroll's answer is to make retention a plan feature: the default keeps sessions for the thirty-day window the remover implements, and longer windows are available to customers who pay for the extra storage and the extra exposure. The remover job itself never changes; only the cutoff moves. That is the clean way to build a retention policy: decide the product question — how long do we promise to remember? — and then make the machinery trivial enough that the answer is just a number.

It is worth noting what the remover is *not*. It is not a MongoDB TTL index, and it could not be. TTL indexes delete documents from MongoDB on a schedule, but the expensive content here lives in the object store, and deleting it requires listing keys, calling the storage API, and updating the session document to `REMOVED` — a cross-store operation no database trigger can perform. A scheduled job with a lock is the honest mechanism for cleanup that touches two systems, and it has a side benefit: the job can delete in batches, survive failures by retrying stale claims, and leave the `REMOVED` tombstone behind as a durable answer to "what happened to this session?" A TTL index would have simply made the row vanish and left the archive orphaned behind it.

The cleanup dance, and why jobs need locks

The remover's deletion is not safe by default. Multiple copies of the job could run at once; the processor might be reading a session's archive at the same moment the remover deletes it; a job could crash halfway through deleting a folder. The remover defends itself with the same **lock pattern** the worker uses, with one constant changed: its claim filter matches `FINISHED` sessions past the retention cutoff whose lock is free or stale, and its lock expiry is longer — about ten minutes rather than two — because deleting every object of a large session takes longer than uploading one does. The claim is again a single atomic `findOneAndUpdate`: whoever stamps the session first owns it; everyone else skips it. A crashed remover leaves a stale lock that expires and lets the next run finish the job.

Why do background jobs need locks at all, when they could each just delete whatever they find? Because every delete-and-update sequence in this system spans two stores and multiple operations. Delete the objects, then mark `REMOVED`: if two jobs run the sequence concurrently, both can list the same objects, both can try to delete them, and neither can tell who won. Worse, without a lock a job could set `REMOVED` on a session whose archive was never deleted, or delete an archive whose status never changed — and the two stores would drift apart

forever. The lock makes the multi-step operation look atomic to the outside world, even though no transaction spans MongoDB and S3. This is the pattern that recurs everywhere in the platform — worker, remover, processor, even small counters — because it is the cheapest way to get coordination between processes that share only a database.

The economics of remembering

Retention policy is ultimately cost policy, so let us do the arithmetic that sits behind the thirty-day decision. The numbers below are illustrative, chosen to be plausible for a small online shop's traffic; your own sessions will differ, and the point is the shape of the math, not the digits.

Consider one session first. A twenty-five-minute visit with steady interaction produces, say, sixty thousand events — mouse movements dominate the count, with mutations, clicks, scrolls, and network entries mixed in. Encoded as protobuf and zipped into frames, that session might occupy a little over a megabyte in the archive. Its MongoDB document, meanwhile, is a couple of kilobytes. Here is the worked example:

Item (illustrative)	Value
Session length	25 minutes
Events recorded	~60,000
Average encoded event	~70 bytes (protobuf, zipped)
Archived size (object store)	~1.3 MiB
Session document (MongoDB)	~2 KiB
Object-store cost per session per month	≈ $0.00003 (at ~$0.02 per GiB-month)

The last row is the surprising one. Storing one session for a month costs about three hundredths of a millicent — effectively nothing. Even multiplied by real traffic, cold replay bytes are cheap, because object storage prices are measured in cents per gigabyte per month, and a session is measured in megabytes. Now scale it up:

Sessions per month (illustrative)	Archived data per month	Steady-state archive (~30-day retention)	Monthly object-store cost
10,000	~13 GiB	~13 GiB	≈ $0.26
50,000	~65 GiB	~65 GiB	≈ $1.30

A few caveats make the picture honest. Real sessions follow a heavy tail: a small fraction of visits — long admin sessions, heavy interactive pages — are ten or fifty times the average, so

plan for the tail, not the mean. Object stores also charge for requests, though reads and writes at this volume are pennies. And the cost table above is only the storage bill; the expensive resources in a session replay platform are the ones this book spent earlier chapters on — bandwidth into the receiver, compute in the worker and processor, and the human time of everyone who watches a replay. Storage is the part of the system where cost scales with *time kept*, which is exactly why retention is the product lever: cutting retention from ninety days to thirty cuts the storage bill to a third and shrinks the privacy exposure window to a third, with no other change to the machinery.

Why thirty days is a sane default

Seen from inside the platform, thirty days looks almost arbitrary; seen from the outside, it is a quietly excellent number. It is long enough to cover the realistic lifecycle of an incident: a bug surfaces, support collects replays over the following days, engineering reproduces and fixes, and the post-mortem wants the original sessions — all of it fits comfortably inside a month. It covers the monthly product rhythm too: the demo for a prospective client, the review of last month's redesign, the training of a new support hire. And it is short enough to keep the promise of data minimization credible, and the storage bill a rounding error, for a company whose traffic spikes with its seasonal calendar.

Thirty days also happens to be the point where the two stores' economics meet. A month of sessions at any small-shop volume is a few dozen gigabytes of object storage — cheap enough that a small customer pays a token amount — while keeping the coordination store light: only a month of live metadata, plus the tiny `REMOVED` tombstones, sits in MongoDB at any time. Longer retention is available to those who need it, and the only thing that changes is the remover's cutoff and the size of the bill. That is the whole argument in one line: the cold archive exists to make remembering cheap, and the remover exists to make forgetting a habit — a habit measured in a month, because a month is long enough to matter and short enough to be affordable, for the platform and for the people whose visits it holds.

> **Story checkpoint — Candlewood Books:** When the free trial ends in July, Tom brings up the plan page and Maya makes the call from across the room: the thirty-day plan. Marta does the math out loud — the double-charge bug, the Safari checkout, the library portal — all of them caught and fixed inside a month, and all of them would still be there to watch a week later if a customer called back with a new detail. Priya notes that a December incident would still be fully visible through the holidays, since the window follows them. For a shop their size the price is pocket change next to one week of support tickets, and Maya would rather renew the promise to their customers than save five euros a month. She sets a calendar reminder to revisit it after the holiday rush.

Chapter takeaways

- The platform divides storage by job: MongoDB is the coordination store for small, hot, queried session documents; S3-compatible object storage is the content store for bulky, cold, read-in-bulk archives.
- The session document is a card-catalog entry — metadata, summaries, and pointers (`storageName`, `s3ServiceName`) to where the content lives — never the log points themselves.
- The lifecycle is a shared state machine with four movers: the receiver keeps sessions fresh, the status job finishes them on silence (two minutes without updates, thirty minutes without interaction, or a three-hour ceiling), the worker archives them, and the remover deletes them.
- Retention is a product decision wearing an engineering costume: the remover's cutoff is a configurable constant, and longer windows change the number, not the machinery.
- Multi-step cleanup across two stores needs the lock pattern; the remover claims sessions with a stale-able lock, deletes the objects, marks `REMOVED`, and unlocks, so crashed runs are retried, not corrupted.
- Cold replay bytes are nearly free; a month of a small shop's sessions costs cents in object storage. Thirty days plus a cold archive is a sane default because it covers the realistic incident lifecycle while keeping storage and privacy exposure bounded.

Chapter 13: The Processor: Turning Replay into Insight

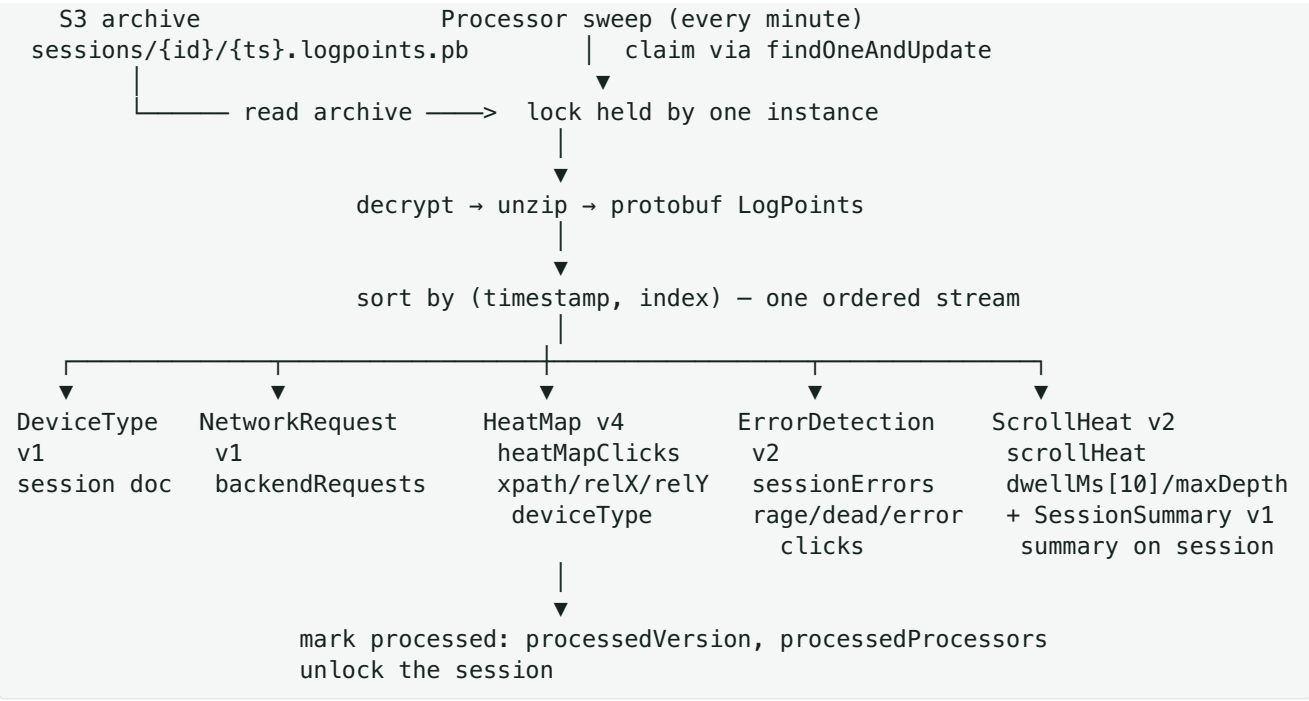
By the time a session is archived, the platform has done the hard work of simply *keeping* it. The recorder captured tens of thousands of events, the receiver encrypted and routed them, and the worker filed the whole timeline into cold storage as chunked, zipped protobuf files. That is the raw material. But a replay of one session answers only the narrowest version of what a support team or product manager is actually asking. Nobody wants to scrub through ten thousand sessions to find the ten that broke. What a session replay platform sells is the ability to ask questions *across* sessions: which pages throw errors, where users click hardest, how far down the page anyone scrolls, which devices are involved. Answering those questions is the job of the **session processor**, and this chapter is about how it works.

The processor is the last major data service in the pipeline. Everything before it moved bytes reliably and cheaply; everything after it reads small, queryable facts. It is the bridge that turns one bulky, encrypted, time-ordered archive into a handful of compact **derived collections** — `sessionErrors`, `backendRequests`, `heatMapClicks`, and `scrollHeat` — plus a few fields written back onto the session document. Replay works without any of that, because the player reads the raw archive directly. Insight does not.

Why the Processor Runs After the Archive

There is a reason processing does not happen on the ingestion path. The analysis needs the *entire* session, in order, decrypted — but a user's session arrives spread across many encrypted network batches, and only the worker knows when it is complete. Processing a half-received session would produce half-truths that would later have to be torn down and rebuilt. Worse, the work is heavy — parsing every event, running windowed detectors, writing thousands of rows — and it must never compete with ingestion for CPU, memory, or database capacity. So LogNroll keeps the hot path thin and runs analysis as a separate, offline consumer of the finished archive.

The eligibility signal is written in the session's status. The worker archives a session and leaves it `FINISHED` with `storageName` set to `SPACES`, meaning its bytes are in object storage. The processor only looks at sessions that reached that state and reads their content from the archive rather than from Mongo: one read of the object-storage files yields the whole decrypted event stream, exactly the unit of work a per-session analysis wants.



There is a second, subtler reason for archive-then-process: it makes the processor *optional and re-runnable*. If the processor is down for an hour, replay keeps working, sessions pile up as `FINISHED`, and the sweep catches up later. If a new version of a detector ships better logic, the platform can re-offer already-processed sessions, and because every analysis step is idempotent, the derived rows converge to the new logic without duplicates or residue. Processing is a projection of the archive, not a step that must happen exactly once at a fixed moment — a property that drives the locking and versioning design the rest of this chapter walks through.

The Claim: Locking a Session for Analysis

LogNroll runs more than one processor instance in production, and nothing tells an instance which sessions to work on. The **claim** pattern — the same one the worker uses for archiving and the remover job uses for deletion — solves coordination without a queue. The `sessions` collection itself is the work queue: every minute each instance runs a sweep that tries to lock the next eligible session, and Mongo's atomic `findOneAndUpdate` guarantees that only one instance can win a given session.

```
Bson filter = Filters.and(
        Filters.eq("status", "FINISHED"),
        Filters.eq("storageName", "SPACES"),
        Filters.gte("startTime", Date.from(Instant.now().minus(14, ChronoUnit.DAYS))),
        Filters.or(
                Filters.exists("processedVersion", false),
                Filters.lt("processedVersion", sessionProcessorPipeline.getProcessVersion(
))),
        Filters.or(
                Filters.eq("lockedBy", null),
                Filters.exists("lockedBy", false),
```

```
                Filters.lt("lockTimestamp", tenMinutesAgoString)));

Document lockedJob = collection.findOneAndUpdate(filter,
        Updates.combine(Updates.set("lockedBy", instanceId),
                Updates.set("lockTimestamp", Instant.now().toString())),
        new FindOneAndUpdateOptions().returnDocument(ReturnDocument.AFTER));
```

The filter encodes the whole contract in one document: the session must be finished and archived; it must have started within the last two weeks (the sweep is deliberately bounded to recent history — sessions older than that were processed long ago or are near their 30-day retention deadline, so re-running them is not worth the work); it must not yet carry the current processing version; and it must be either unlocked or locked by an instance whose **lock lease** has expired. Each instance draws a random `instanceId` at startup and stamps it into every lock it takes. The lease is the crash-recovery safety net: if an instance dies mid-session, its locks grow stale, and after ten minutes another instance may take over. A stale lock is not a bug; it is the system's way of declaring a previous worker dead and moving on.

The sweep runs on a fixed schedule — a scheduled task with a one-minute rate and a short initial delay — and within one tick an instance claims up to twenty sessions by default, configurable through the `PROCESSOR_MAX_SESSIONS_PER_RUN` environment variable (the `processor.max-sessions-per-run` setting, default 20). The bound exists for a practical reason the code comments make explicit: when the processor falls behind and starts catching up, an unbounded run would load dozens of whole decrypted sessions into memory at once and spike the heap. Twenty sessions per minute per instance keeps the backlog draining in controlled slices. If a session cannot be processed — a corrupt archive, a parsing exception — the instance unlocks it and moves on; a permanently broken session stays unprocessed rather than blocking the queue. Out-of-memory errors are treated differently: the instance unlocks the session and rethrows, letting the container restart, because an OOM during one session is a sign that the whole process is unhealthy.

The claim also explains the graceful-shutdown dance in the service's lifecycle code. When Kubernetes wants to stop a pod, the processor signals that shutdown is starting, waits for the current tick's work to finish (up to a couple of minutes), and only then exits. A claimed, half-processed session must not be abandoned with a fresh lock that looks alive; the shutdown window lets the instance finish or leave the lock to age out naturally.

From Archive to Ordered Events

Once a session is claimed, the processor loads its archive through the same two object-storage tenants the worker writes to (the default Spaces service and the POLISUA tenant, selected per session by the session's `s3ServiceName`). It reads the chunked zip files, decrypts each event payload with the shared AES key, and decodes the protobuf `LogPoints` batches back into individual `LogPoint` records.

Then comes a step that is easy to underrate: the pipeline sorts the stream by `timestamp`, then by `index`. The recorder batches events and ships them over the network, so the on-disk order is roughly, but not exactly, chronological. Nearly every analysis step cares about strict order: a dead-click detector needs to know whether a DOM mutation arrived after a click, and a scroll-attention accumulator needs to know which scroll position came before which. Sorting once, up front, turns a bag of batches into a timeline that every processor can walk in a single pass. The pipeline hands that ordered list to each step in turn, and each step returns the version it applied.

The Processor Walk

LogNroll implements each analysis as a small, independently versioned processor bean: `DeviceTypeSessionProcessor`, `NetworkRequestSessionProcessor`, `HeatMapSessionProcessor`, `ErrorDetectionSessionProcessor`, `ScrollHeatSessionProcessor`, and `SessionSummaryProcessor`. Each has a stable name, a declared version, and an `@Order` that fixes the sequence. The pipeline sorts the events once, then lets each step consume the same stream. This modularity is what makes the version gate in the next section possible: a step can be re-run alone, at a new version, without touching its neighbors' output.

Step 1: Error Detection

The first derived collection, `sessionErrors`, most directly feeds support and the daily error digest. `ErrorDetectionSessionProcessor` listens for two kinds of events. Console errors arrive as `LOG` events whose payload marks them as `error` level — the uncaught exceptions and failed assertions the SDK's console wrapper captured. Network failures arrive as `NETWORK` events whose payload carries an error stage, an error object, or an HTTP status of 400 or higher. Each becomes a `SessionError` document carrying the session, company, and user ids, the page URL, the console or HTTP method, the HTTP status when there is one, the message, and the event timestamp.

The interesting part is what the processor adds on top: three **behavioral issues** derived purely from events the recorder already captures, so no SDK change was ever needed. A **rage click** is three or more clicks on the same element within two seconds — the canonical "I clicked and nothing happened, so I clicked again" pattern. A **dead click** is a click on an interactive element (an anchor, button, input, select, textarea, label, or summary) after which the page visibly does nothing: no DOM mutation, no navigation, no input or form event within two seconds. Clicks near the very end of the session are excluded, because closing the tab right after clicking is not a failed interaction, and so are clicks better explained as rage or error clicks. An **error click** is a click followed within three seconds by a console error or a failed request — the moment the user's action broke something. The detectors run in a

deliberate order — rage first, then error clicks, then dead clicks — so one frustrated burst yields one rage-click error rather than a pile of overlapping diagnoses.

Every error gets a **fingerprint**: the message is normalized (UUIDs and long number sequences become wildcard tokens, because a real order id or session token should not split otherwise identical errors into separate groups) and hashed with SHA-256, truncated to a readable length. Two users hitting the same bug on the same page produce the same fingerprint even when their order ids differ, which is exactly what lets the error digest count occurrences and show sample sessions. Compound indexes on company plus fingerprint and company plus timestamp make both reads cheap.

Step 2: Network and Backend Analysis

The network step parses the same `NETWORK` events and flattens each into a row in `backendRequests`: the HTTP method, the request URL, the parsed host, the decoded query string, the response code, and the duration measured from the request and response timestamps (or from the response's reported time when only that is available). Requests that never got a response carry code 0. These rows are the structured form of the replay's network timeline, and a dedicated analysis service turns them into the numbers a dashboard shows — total, successful, and failed counts, success rate, average and maximum response time, top hosts, slowest requests, method and status distributions — per session or per company. Where error detection asks "did this request break the user's journey?", the network step asks "what are this company's backends doing in aggregate?"

Step 3: The Click Heatmap, xpath Identity and All

`HeatMapSessionProcessor`, currently at version 4, converts `CLICK` events into `heatMapClicks` rows. The recorder's click payload carries `x, y, lnrId, tagName, relX, relY, xpath`, and the processor's first decision is a filtering one: only clicks that carry an element `xpath` are kept. That rule is the heart of the design, and it is a classic subtle bug worth stating plainly. `lnrId` is a *per-session* counter that the recorder assigns to elements for its own DOM matching; the same button has lnrId 17 in one session and lnrId 83 in another, so aggregating by it would scatter the button's clicks across hundreds of meaningless groups. The **xpath** is the only stable identity that survives across sessions — the path from the document root to the element, identical in every session that renders the same button — so it is the aggregation key, and `lnrId` is stored only as a per-session reference. Clicks without an xpath are dropped, because a click that cannot be attached to a stable element cannot be compared across sessions.

Version 4 added two refinements. First, the processor stores `relX` and `relY` — the click's coordinates relative to the clicked element's border box. When the heatmap renderer later

draws a click, it resolves the xpath, finds the element's current position, and places the heat at `element.left + relX` instead of falling back to the element's center, so the click keeps its original precision even after the page has been re-laid-out. Payloads from older recorders, lacking the relative fields, are parsed through the legacy format. Second, every click is classified into a **deviceType** bucket — mobile, tablet, or desktop — using the user agent and viewport width from the latest `META` event seen up to that point, with the session's recorded OS and browser as fallback. Classifying per click rather than once per session matters because responsive pages change layout mid-session: the same visitor may be desktop-width at noon and phone-width after resizing, and their clicks belong in different buckets. The stored URL is stripped of its query string and fragment so tracking parameters do not fragment one page into thousands of pseudo-pages. Each row carries the session, company, and user ids, the tag name, and version 4, and the compound index on company, URL, and device type is exactly the lookup path the heatmap endpoints use.

Step 4: Scroll Heat, Version 2

`ScrollHeatSessionProcessor` answers a question that plain analytics dashboards get wrong: not "how far did people scroll?" but "how long did each part of the page actually stay in front of them?" A maximum-scroll-depth number is misleading on dynamic pages, where the document height changes as content loads, and it says nothing about whether a section held attention. Version 2 replaced depth-only entries with **time-weighted dwell**.

The processor watches window `SCROLL` events, whose payloads carry the absolute scroll position `y` and the scrollable range `maxY`. Element-internal scrolls (reported with a target element marker) are ignored, as are pages that are not scrollable at all. For each session and URL it keeps an accumulator and, between two scroll events, attributes the elapsed time to the ten depth bands — each covering 10 percent of the page's depth — that the viewport actually covered along the scroll path. A band counts as visible while the viewport window overlaps it; the math interpolates the fraction of time each band was on screen for a moving scroll and counts a stationary viewport as fully holding every band it shows. The viewport height comes from `META` events, because the same bands cover different amounts of screen on different devices.

Two details keep the numbers honest. First, the **idle cap**: intervals longer than 30 seconds between scroll events are treated as inactivity and never attributed, so a scroll-then-coffee-break must not inflate the dwell of whatever happened to be on screen. Second, re-normalization: `maxDepth` is recomputed at every event as `y / maxY` of the *current* scrollable range, so it stays a meaningful fraction even as the page grows. When a navigation or the end of the session closes a page's segment, the remaining interval is attributed only up to that 30-second horizon, and pages with no measurable scroll activity produce no row at all. The resulting `scrollHeat` document stores `dwellMs` — a ten-element array, one entry per band —

plus `maxDepth`, `totalDwellMs`, the `viewportHeight` used in the math, and the `deviceType` bucket, classified as clicks are. Version 1 rows wrote a different shape, which is why version 2 bumps the processor version (next section) and why readers filter on `version >= 2`: mixing the old depth-only shape with the new dwell shape would silently corrupt aggregates.

Step 5: Device Classification and the Session Summary

Two smaller steps round out the walk. `DeviceTypeSessionProcessor` classifies the whole session once — mobile, tablet, or desktop — using the same classifier the heatmap and scroll steps use per event, and writes the result as `deviceType` on the session document so the dashboard can filter session lists by device without touching the archive. The classifier is pragmatic and rule-based: tablet markers like iPad, tablet, Kindle, Silk, and PlayBook; phone markers like iPhone, iPod, Windows Phone, BlackBerry, and Android-with-mobile; and a viewport-width fallback (below 768 pixels is mobile, below 1024 is tablet) for what string matching misses. Finally, `SessionSummaryProcessor` builds a compact, deterministic digest in a single pass — the page journey with per-page dwell times, interaction counts, an error digest, and network statistics such as average response time and top failed endpoints — and stores it as a `summary` field on the session document with a targeted update that never touches fields a user may have edited. Behavioral issues are merged in from the rows `ErrorDetectionSessionProcessor` already wrote, so the summary reflects them without re-running detection. The digest is deliberately cheap: it is what a session list shows before anyone opens a replay, and it costs zero AI tokens. Best-effort by design, it is the one step that may fail silently rather than fail the pipeline.

The Version Gate: PROCESS_VERSION and the Reprocessing Gotcha

The versioning scheme that runs through all of this is the single most instructive design detail in the processor, because it is both elegant and easy to get wrong. Every step declares a `PROCESS_VERSION` — today NetworkRequest and DeviceType at 1, ErrorDetection and ScrollHeat at 2, HeatMap at 4, SessionSummary at 1 — and after a successful run the session records two things: a coarse integer `processedVersion` and a fine-grained map, `processedProcessors`, of step name to the version that step applied. The coarse integer answers the database filter cheaply — is this session behind the pipeline's current total? — and the fine map decides which steps actually need to run:

```
int currentVersion = applied.getOrDefault(processor.getName(), 0);
if (currentVersion >= processor.getProcessVersion()) {
    // already applied at this version — nothing to do
    continue;
}
```

```
int version = processor.process(session, orderedPoints);
applied.put(processor.getName(), version);
```

The rule is `applied >= current means skip`. And here is the gotcha the code comments warn about in so many words: if you change what a step's output *means* or what shape it takes, and you forget to bump that step's `PROCESS_VERSION`, the pipeline silently skips every session that already ran — the stored applied version is still equal to the current one, so the sweep never offers the session, the step never runs, and the derived rows keep their old, now-wrong semantics forever, with no error anywhere. Version bumps are not optional ceremony; they are the mechanism by which changed logic propagates to already-processed sessions.

When a version does bump, the machinery works in your favor. When HeatMap moved up to its current version 4, the pipeline's total rose with it, so every previously processed session became eligible again at the sweep. The fine map then did the real bookkeeping: only the HeatMap step re-ran, because every other step's applied version still met the `applied >= current` test. Old heatmap rows were deleted and rewritten in the new shape; no other collection was touched, no session was double-processed, and the fleet converged in the background. Legacy sessions recorded before the fine map existed are handled by reconstruction: their old integer is decoded into the historical pipeline it implies (NetworkRequest and HeatMap at version 1), so newly added steps such as ErrorDetection run for them without re-running the older ones.

The version field is also stamped onto every derived row — the *reader's* half of the contract. When a collection's shape changes, consumers gate on the stored version, as the scroll-heat readers do with `version >= 2`, so a mix of old and new rows can never poison an aggregate while reprocessing is still catching up. Writers bump versions and readers check them, which makes schema evolution a background event rather than a coordinated deployment.

Derived Collections as Product Data

Stepping back, the derived collections exist to convert the archive's event model into the product's query model. An event stream is ordered and linear; it answers "what happened next?" Product questions are relational: "show me every occurrence of this error fingerprint this week," "give me clicks on this URL for mobile devices," "aggregate dwell per band for this page." Each derived collection is indexed for exactly those questions — `sessionErrors` on company with fingerprint and with timestamp, `heatMapClicks` on company, URL, and device type, `scrollHeat` on company, URL, device type, and version. The daily error digest groups `sessionErrors` by fingerprint and pulls sample session ids; the heatmap and scroll endpoints page over the two heat collections; the network analysis service reads `backendRequests`; even the AI-assisted analysis layer in the platform's MCP server reads these collections directly rather than re-decrypting archives. A few hundred compact documents per session stand in

for tens of thousands of events, and that difference in query cost is the difference between a feature that ships and a feature that only works in a demo.

Idempotency and Crash Recovery

Every step in the walk is written to be safe to run twice. Each processor begins by deleting its own rows for the session id and then inserts fresh ones, stamping every row with its version; no step appends blindly to what a previous run left behind. That single habit — delete-then-rewrite, scoped per session — makes the whole pipeline **idempotent**, and idempotency is what makes crash recovery a non-event. If an instance dies after writing derived rows but before marking the session processed, the lock goes stale and another instance claims the session after ten minutes; reprocessing deletes the half-written rows and rewrites them, converging to the same correct state. If it dies after marking processed, there is nothing to redo. If two instances ever raced for the same session, the atomic claim ensures only one wins. No multi-document transactions are needed across these Mongo writes, because re-running a step is equivalent to running it once: the lock makes work exclusive, idempotency makes it repeatable, and the lease makes abandoned work recoverable.

That is the quiet triumph of the design: distributed processing semantics without a distributed system. Sessions are the queue, claims are the scheduling, leases are the timeout, and versions are the migration tool. The discipline it demands of every future processor author is small — never change output shape without bumping the version, never write a step that cannot safely re-run — and the payoff is a processing layer that catches up after outages, upgrades itself in the background, and never needs a manual reconciliation script.

> **Story checkpoint — Candlewood Books:** On a Thursday in June the 09:00 error digest lands in Priya's inbox with a fingerprint she has never seen: a console error on the checkout page, 40-plus occurrences, with sample session links. She opens one replay from a phone, and there it is — the exact click that did nothing, followed in the timeline by the uncaught error from the stale cached bundle. The processor had flagged the same pattern as error clicks, and the search page had quietly accumulated rage clicks for weeks. The fix ships the same day, the digest goes quiet the next morning, and for the first time since April the support ticket count actually falls. Marta prints the two emails and pins them above her desk: the system had seen the whole story before any human asked.

Chapter takeaways

- Processing runs off the hot path, after archiving, because whole ordered sessions only exist in the archive and analysis must never compete with ingestion.

- The `sessions` collection is the work queue: an atomic `findOneAndUpdate` claim with an instance id and a ten-minute lock lease lets many processor instances divide work safely and recover from crashes.
- The processor walk turns one ordered event stream into compact derived collections: `sessionErrors` (including rage, dead, and error clicks), `backendRequests`, `heatMapClicks`, and `scrollHeat`.
- Heatmap identity is xpath, never `lnrId`: `lnrId` is a per-session counter, while xpath is the only key stable across sessions; `relX`/`relY` preserve click precision, and per-click `deviceType` keeps responsive sessions in the right buckets.
- Scroll heat v2 measures time-weighted dwell per 10 percent band with viewport-height interpolation and a 30-second idle cap, and stamps rows with the processor version so readers can filter old shapes out.
- `PROCESS_VERSION` is the migration mechanism: bump it when a step's output shape or semantics change, or the pipeline silently skips reprocessing (`applied >= current`); writers bump versions and readers gate on them.
- Delete-then-rewrite per session makes every step idempotent, so crashes, stale locks, and version upgrades all converge without manual cleanup.

Chapter 14: Background Jobs, Monitoring, and Alerts

A session replay platform is full of work that nobody triggers by hand. Sessions have to time out when users vanish mid-visit; archives have to be deleted when their 30-day retention window closes; hosts have to be probed, certificates checked, and emails sent when things break. None of this can wait for an engineer to notice. This chapter covers the quiet machinery that keeps the lifecycle honest — the two scheduled jobs that move sessions between states — and then the bonus product feature that the LogNroll team built on top of the same data: a host-monitoring watchtower that turns recorded session URLs into uptime and TLS alerts for the sites its customers actually run.

The Lifecycle Needs Janitors

Recall the session state machine from earlier chapters: `ACTIVE → IDLE → FINISHED → REMOVED`, advanced by whoever is in a position to know the truth at that moment. The receiver flips a session back to `ACTIVE` while batches are still arriving and marks it `IDLE` when interaction stops. The worker archives finished sessions into object storage. But two transitions have no natural trigger. A session can sit `IDLE` forever if its user simply closed the laptop — nothing will ever arrive to finish it. And a `FINISHED` session can sit in object storage forever, costing money and outliving its purpose. Those two transitions are the job of dedicated background jobs, both small Go programs that run as Kubernetes CronJobs against the shared `sessions` collection, and both written to be safe to run again at any time.

The Every-Minute Status Job

The first job, `lognroll-session-status-job`, is scheduled for every minute of every hour. It does one thing: find `ACTIVE` and `IDLE` sessions that should no longer be considered alive, and mark them `FINISHED`. A session finishes when any of these hold:

```
status-job (every minute):
  lastInteraction older than 30 minutes        → FINISHED
  no lastInteraction, started > 30 minutes ago  → FINISHED
  lastUpdate older than 2 minutes               → FINISHED
  no lastUpdate, started > 30 minutes ago       → FINISHED
  started more than 3 hours ago (max session)   → FINISHED
```

The thresholds are deliberately layered. The 2-minute no-update rule catches sessions whose recorder stopped sending anything (the browser tab died, the network dropped); the 30-minute idle rule catches users who walked away; the 3-hour maximum is a backstop that guarantees no session can live longer than a hard ceiling, whatever combination of events

and missing events the data shows. When a session is finished, the job stamps `endTime`, computes `duration` from the last interaction (or zero if there never was one), and writes `status = FINISHED` with a targeted update. There is no lock dance here, and none is needed: the work is idempotent by construction, because a session that is already `FINISHED` no longer matches the query. If a run overlaps itself, or two pods of the job ever ran at once, the second run simply finds nothing left to do. This is also the moment in the lifecycle that hands off to the rest of the pipeline: only a `FINISHED` session can be archived by the worker and later claimed by the processor for analysis.

The Hourly Remover Job

The second job, `lognroll-session-remover-job`, enforces the platform's 30-day retention promise. Retention is a product decision — Candlewood picked the 30-day plan, and 30 days is the default the code encodes — so the job must be merciless about what it does at that boundary. It runs hourly in production, claims every `FINISHED` session whose `startTime` is older than 30 days, and removes its data permanently.

Because deletion is destructive, the remover uses the same claim pattern you saw in the worker and the processor: an atomic `findOneAndUpdate` that only wins when the session is unlocked or its lock is stale (again, ten minutes), stamped with a remover-specific instance id. Only one remover run can delete a given session. The job then lists every object under the session's archive prefix — `sessions/{sessionId}/` — and deletes them in batches through the object-storage API, selecting between the two storage tenants the way every other service does, via the session's recorded `s3ServiceName`. Only after the objects are confirmed gone does it set `status = REMOVED` and unlock the session, looping until no eligible sessions remain. If a deletion fails, the session is unlocked and retried on the next hourly run rather than being half-deleted and abandoned. Each deletion is followed by an explicit garbage-collection hint, a small habit the Go port carried over from the Java implementation to keep the job's own memory footprint flat while it chews through a long backlog after, say, a retention change. This job replaced the Java archiver's removal half, and together with the status job it completes the split that gave the platform two small, single-purpose CronJobs instead of one large scheduled service.

The Watchtower: Monitoring Built on Session Data

The two jobs above are plumbing. The monitoring module inside `lognroll-app-api` is the part that makes you stop and think: the platform already records every URL a user visits in every session, so why not use that data to watch over the hosts those URLs point at? That is exactly what LogNroll did. The module maintains three Mongo collections: `hostMonitors`, one document per monitored origin per company; `hostUptimeSamples`, a history of probe

results; and `hostAlerts`, a log of every incident. A monitored host is an origin like `https://www.candlewood.example`, with its own state: whether it was discovered or added manually, whether it is enabled, its current uptime status (`UNKNOWN`, `UP`, `DOWN`, or `PAUSED`), its certificate status, how many sessions it has been seen in, and internal bookkeeping flags that guarantee alerts fire exactly once per incident.

What makes the module a *watchtower* rather than a toy uptime checker is that it runs itself mostly hands-free, and it is careful about the two failure modes that would make it annoying: noise and surprise.

Discovery: Reading URLs You Already Record

A daily job at 03:30 scans the `urls` arrays on recent sessions across all companies and extracts candidate hosts. For each session URL the discovery service normalizes it down to a clean origin — scheme plus lowercased host plus an explicit port only when it is not the default — and filters hard. Hosts like `localhost`, `.local`, `.lan`, `.internal`, loopback, private, link-local, and multicast addresses are excluded outright, because monitoring your own laptop from a multi-tenant SaaS is neither useful nor sane. Each origin counts once per session, no matter how many pages of it the user visited, and only origins seen in at least three sessions become candidates, which filters out one-off typos, dev machines, and third-party redirects that a single user happened to hit. Candidates are capped per company to guard against a junk flood.

Then comes the design decision that shows real product discipline: **candidates are created disabled**. Discovery writes a `hostMonitors` document with `source = DISCOVERED`, `enabled = false`, and status `UNKNOWN`, and it never touches an existing monitor's settings. The platform will happily suggest hosts to you, but it will not start probing them, spending its own resources on them, or emailing you about them until a human opens the monitoring tab and activates the ones that matter. No auto-enable, no silent overwrite of a user's configuration, ever. If a host disappears from recent sessions, the module does not delete it on a whim; a stale-host scan at 04:45 merely flags hosts with no sessions in the last seven days and lets the operator pause or remove them, because a long-running campaign page might legitimately go quiet without becoming irrelevant.

Probes: HEAD First, GET When Needed

An uptime checker runs on a 60-second scheduler tick and probes whichever enabled hosts are due. Each host has a probe interval that defaults to 300 seconds — five minutes — so a company with a dozen hosts costs the platform a steady trickle of requests, not a storm. The probe itself is a study in real-world HTTP: it sends a `HEAD` request first, and a `2xx` or `3xx` response counts as up. But many servers answer `403` or `404` to `HEAD` while serving `GET`

happily, and some web application firewalls reject non-GET methods outright. So a non-successful `HEAD`, or a transport-level failure, triggers a `GET` fallback on the same URL, with a browser-like User-Agent to reduce the chance of tripping bot filters. Only when the `GET` also fails is the host reported down. The TLS handshake for probing uses a trust-all context deliberately: availability is judged by reachability, so a host with a private or otherwise untrusted-but-working certificate still counts as up — certificate health is the certificate checker's job, reported separately, and hostname verification stays on so a genuinely mismatched certificate still fails the probe.

Every probe writes one sample into `hostUptimeSamples` with a timestamp and the measured response time. The samples collection carries a TTL index that automatically drops documents after 30 days, which conveniently matches the longest statistics window the dashboard charts — the module keeps a month of raw history and no more. Status transitions are conservative. A single failed probe does not flip a host to `DOWN`; the checker requires three consecutive failures before declaring an outage, so transient blips — a network hiccup, a deploy — show up in the samples without firing alerts or flapping the status. When the third consecutive failure lands, the host's status becomes `DOWN` and an incident begins.

Certificate Checks: Daily TLS Health

Separately, a daily job at 04:15 checks the TLS certificates of all enabled HTTPS hosts. It opens a raw TLS handshake with a trust-all context, reads the presented leaf certificate, and classifies its remaining lifetime: expired, expiring within the renewal window (30 days by default), or valid. The result is stored on the monitor — issuer, expiry date, days left, status — and shown in the dashboard's host table, so a certificate problem is visible for weeks before it becomes an outage instead of surfacing only when a browser starts showing scary warnings.

One Alert per Incident

Both checkers are built around the same principle: fire exactly once per incident, and clear when it is over. Internal flags track whether the current outage or certificate problem has already been alerted. When a host flips `UP → DOWN`, the module records a `DOWNTIME_STARTED` event in `hostAlerts` and sends one down email; further failed probes stay silent. When the host recovers, a `DOWNTIME_RECOVERED` event and a recovery email go out, and only if a down alert was actually sent earlier. Certificates work the same way: entering the expiring or expired state sends one alert, and observing a renewed, valid certificate clears the flag so the next expiration can alert again. Every incident also lands in `hostAlerts`, giving each company a scrollable history of its outages, recoveries, and certificate warnings.

Deleting a Host Must Win Every Race

The most quietly important detail in the module is how checkers persist their results. Both the uptime checker and the certificate checker save their state with a *conditional update* filtered on `_id` plus `companyId`, never a whole-document `save()`. Consider the race: a probe starts, the user deletes the host in the dashboard, the probe's HTTP request completes a moment later, and the checker writes its result. With a naive save, that late write would recreate a document the user just deleted — the monitoring module would resurrect a host they explicitly removed, and start emailing them about it again. With the conditional update, the write matches zero documents and silently does nothing. Deletion always wins, and the checkers add a belt-and-suspenders existence check before persisting for the same reason. This is the kind of bug that only shows up under production load, and it is worth internalizing: any background writer that races with user deletes should update conditionally rather than save unconditionally.

The module's API mirrors all of this: under `/api/companies/{companyId}/monitoring/`, a company can list, create, edit, and delete hosts, request an on-demand probe, page through samples and alerts, and browse discovery candidates — including a refresh action that re-scans sessions and an activate action that enables a candidate. Uptime percentage and average response time are computed from the samples for the 24-hour, 7-day, and 30-day windows the dashboard charts show.

Alert Emails and the Error Digest

All of this alerting funnels into email, which the platform renders through a small dedicated service rather than hand-building HTML in each job. `lognroll-mailtemplate` is an Express application that composes transactional mail with pug templates and the MJML framework and exposes one endpoint per email type. The monitoring module calls `/host-down`, `/host-recovered`, and `/cert-expiry`; a separate job renders the daily error digest through `/error-alert`. The app's mail component then delivers the rendered HTML through its transactional email provider.

The recipients are always the company's own team. For monitoring alerts, the sender resolves the company's active team members to their email addresses, then checks three gates before sending: the global `monitoring.alert-email-enabled` switch, the per-host `alertEnabled` flag, and the company-level monitoring-alert toggle that lives in the settings page. Every gate defaults to on, but every one of them exists because a product that emails a sleeping founder at 3:00 a.m. about a host they stopped caring about has thirty seconds to lose a customer.

The daily error digest deserves special mention, because it is the alert that connects this chapter back to the previous one. Every day at 09:00, `ErrorAlertJob` looks at each company's `sessionErrors` from the last 24 hours, aggregates them by fingerprint, and sends one email listing the top problems — type, message, occurrence count, and links to sample sessions in the player. When the processor's error detection catches a new bug and the digest renders it the next morning, the company's whole team sees it before a single customer complains. For companies that have switched the digest on, this single email replaces the ritual of manually watching the error list, and it is precisely the email that Candlewood Books' support team learned to read first.

```
monitoring.alerts gate (in order):
  global monitoring.alert-email-enabled   (default true)
  per-host alertEnabled                   (default true)
  company monitoringAlertEmailEnabled     (default true)
  → sendHostDownAlert / sendHostRecoveredAlert / sendCertExpiryAlert
daily error digest (09:00), per company with errorAlertEmailEnabled:
  sessionErrors (24h) → group by fingerprint → /error-alert email
  with sample session links into the player
```

The Product Lesson: Replay Watches the Hosts Your Users Actually Use

Stepping back, the monitoring module is a small illustration of something larger about session replay as a product. Every other monitoring tool on the market makes you tell it what to watch: you type in a URL, choose a frequency, configure an alert. LogNroll's module flips that around. The recorder already knows what your users actually visit, because it records every navigation. Discovery simply reads that data back, normalizes it, filters out nonsense, and offers you the result as one-click candidates. The hosts that matter are, almost by definition, the hosts your sessions keep returning to — and a shared host that serves your storefront but is dying under someone else's traffic is exactly the kind of thing you would never think to monitor until it takes you down.

The design choices along the way are lessons you can carry into any system that generates alerts from user data: prefer candidates over assumptions, so the machine suggests and the human decides; make noise expensive, by requiring consecutive failures before declaring an outage and by alerting once per incident rather than once per check; keep raw history (the samples) separate from state (the monitor document) so every claim can be audited; and make background writers that race with deletions lose that race by updating conditionally. A watchtower is only useful if it is quiet when nothing is wrong and unmistakable when something is.

> **Story checkpoint — Candlewood Books:** At 3:12 a.m. on the first Saturday of the summer promo, Tom Bakker's phone lights up with a LogNroll host-down email: the storefront origin is DOWN. His first thought is the CDN — but the CDN is fine, still serving cached pages, which is exactly why nobody noticed during the day. The checkout origin, the part that has to hit the server, has been failing for three consecutive probes. Tom logs in from bed, checks the samples chart, and sees the failure curve start right after midnight. Their shared host — cheap, shared, and quietly dying under another tenant's load — is the culprit. He moves the origin to the backup server by 6:00 a.m., and the recovery email arrives before the first Saturday shoppers finish their coffee. Nobody at Candlewood had ever thought to monitor the host; the platform had simply watched where their sessions pointed and offered to keep an eye on it.

Chapter takeaways

- The session lifecycle is kept honest by two small CronJobs: the every-minute status job marks stale `ACTIVE` and `IDLE` sessions `FINISHED` (30-minute idle, 2-minute no-update, and 3-hour maximum rules), and the hourly remover job deletes archives older than 30 days.
- The remover reuses the claim pattern — atomic lock with a ten-minute stale lease — because deletion is destructive, while the status job needs no lock because its work is idempotent.
- Host monitoring is a watchtower built on data the platform already records: a nightly discovery job normalizes `sessions.urls` into origins, filters out local and private hosts, and creates disabled candidates that humans activate.
- Probes run HEAD first with a GET fallback (403/404 HEADs are common), default to one probe per host every five minutes, and only declare `DOWN` after three consecutive failures.
- A daily certificate check classifies every enabled HTTPS host as `VALID`, `EXPIRING` (within 30 days), or `EXPIRED`, and both checkers alert exactly once per incident, clearing on recovery or renewal.
- Checkers persist through conditional updates on `_id` plus `companyId`, never `save()`, so a probe that races a host deletion can never resurrect it.
- Alert emails and the daily 09:00 error digest are rendered by a dedicated mailtemplate service (`/host-down`, `/host-recovered`, `/cert-expiry`, `/error-alert`), gated by global, per-host, and per-company switches that default to on.

Chapter 15: Serving a Session: The Player API

Every byte the platform collects — every click, keystroke, and DOM mutation from the earlier chapters — is worthless until a human can watch it. The player API is the read side of that bargain. Where the worker files sessions into the cold archive and the session processor mines them for insight, the player API does one narrow thing: when a member of your company asks to open a session, it proves they are allowed to, fetches the archived events from object storage, decrypts them, puts them back in order, and hands the browser a single binary protobuf payload that the player frontend turns into a replay.

Notice how small that job is. The API never renders a pixel, never rebuilds a DOM, never runs the recorded page. Rendering is a browser problem, and Chapter 16 covers it in full. This chapter is about the contract between storage and screen: authentication, authorization, retrieval, decryption, and ordering.

Where the player API sits

The pipeline should feel familiar by now: a recorder captures events and posts them to the receiver, which encrypts each payload and drops the batch onto a NATS subject; the worker pulls the subject, decrypts and re-chunks the stream, and uploads per-timestamp zip archives to object storage under a `sessions/{sessionId}/` prefix; the session processor reads the same archive to derive errors, network analysis, and heat data. The player API is a third reader of that archive, sharing almost everything with the processor: the same Mongo `sessions` collection, storage tenants, AES key, and protobuf contract.

The asymmetry is deliberate. Everything before this stage is a write path with few producers; the player API is a pure read path with many consumers. A replay can be opened from the dashboard, deep-linked from a chat message, or loaded three times in a row by the same person hunting for the exact second something broke. Each read must be fast, because nobody waits happily; correct, because a replay that shows events out of order is worse than none; and safe, because you are streaming one stranger's private browsing history to someone who must be allowed to see it.

Who may watch: the token, the handoff, and company scoping

Access control starts long before the player API sees a request. When a user logs into the LogNroll dashboard, the app API issues a JSON Web Token signed with a shared HS256 secret. The same secret is known to the app frontend, the player frontend, and the player API, so a token minted by the control plane is accepted by the replay service without a second

round trip. The token carries the user's email as a claim, and that claim anchors every authorization check downstream.

The awkward part is that the dashboard and the player live on different origins, and browsers do not share storage between origins. So when the app frontend opens the player at `/companies/{companyId}/session/{sessionId}`, it appends the JWT to the URL as a query parameter named `lr_token`. This is the cross-origin auth handoff, treated as a hot potato. On startup the player's auth service reads the parameter, verifies the token is not expired and carries an email claim, persists the token into the player origin's own storage, and immediately rewrites the URL with `history.replaceState` so the token vanishes from the address bar and cannot leak through referrer headers. An expired or malformed handoff is dropped and never persisted — the code is explicit that a stale token must not clobber a still-valid one, or every API call would fail and leave the user looking stuck.

From then on the player attaches the token itself: an HTTP interceptor adds an `Authorization: Bearer <token>` header to every outbound call, including the calls back to the app API for session metadata and error lists. A 401 in response marks the stored session as dead: the interceptor clears it and redirects to login. The player also supports logging in directly through the same magic-link flow the dashboard uses, so the replay page works standalone as well as handed off.

```
// JWT middleware in the player API (Go, abridged): parse the bearer token
// with the shared HS256 secret and put the email claim into the request context.
authHeader := r.Header.Get("Authorization")
if authHeader == "" {
    http.Error(w, "Authorization header required", http.StatusUnauthorized)
    return
}
parts := strings.Split(authHeader, " ")
if len(parts) != 2 || parts[0] != "Bearer" {
    http.Error(w, "Invalid authorization header format", http.StatusUnauthorized)
    return
}
parsed, err := jwt.Parse([]byte(parts[1]), jwt.WithKey(jwa.HS256, m.secret))
if err != nil {
    http.Error(w, "Invalid token", http.StatusUnauthorized)
    return
}
```

Authentication proves who you are; authorization proves you may look at this session, and LogNroll scopes that check to the **company**, not the session. Every session document records the `companyId` it was recorded under. To serve a session, a handler first resolves the email claim to a user document, then asks the team membership collection whether that user holds an active membership in the session's company. No membership, no data: the API answers 403 with "Not authorized to access this company" and the client receives nothing readable. This is the multi-tenant boundary at work — a valid token for a user in Company A is useless against Company B's sessions, even though both sit in the same Mongo collection

and the same storage buckets. The check repeats on every endpoint, including the heatmap family, where the `companyId` arrives as a query parameter and must likewise belong to the caller.

Finding the session and reading the archive

The main endpoint is `GET /api/get/{id}` under a configurable context path (default `/api`), where `{id}` is the session's Mongo object id. The handler first looks the id up in the shared `sessions` collection; an id that matches nothing produces a 404. A session that exists is checked for access, as above, and marked watched as a best-effort side effect so the session list can distinguish "never opened" from "seen."

Two fields on the session document tell the API how to fetch events. `storageName` records whether the session was ever archived — only archived sessions carry the value `SPACES`, set by the worker after a successful upload. `s3ServiceName` records which object storage tenant holds the archive, because the platform runs two S3/Space tenants and the API must read from the same one the worker wrote to. Selecting the wrong tenant would be like asking the wrong library branch for a book: the shelf simply is not there.

A session whose `storageName` is not `SPACES` has no archive yet. It may still be ACTIVE or IDLE with events waiting on the bus for the worker, or it may have failed archiving. In either case the API returns a valid, empty `LogPoints` payload with HTTP 200 rather than an error — nothing was archived, so there is nothing to serve, and an empty-but-successful reply lets the player show a graceful "no session data" state. It is the same reply a session returns after its retention window expires, as the failure section below explains.

For archived sessions, retrieval is a small exercise in object storage. The API lists every object under the `sessions/{sessionId}/` prefix, sorts the object keys, downloads each one, unzips its single contained file, and concatenates the results into one contiguous byte stream. The sort is not cosmetic: the worker names each chunk after the timestamp of its first frame, roughly `{timestamp}.logpoints.pb`, so lexicographic key order is chronological order and the concatenated stream is already time-ordered before any event is inspected.

```
sessions/<sessionId>/
  1715000000000.logpoints.pb   <- zip #1 (earliest chunk)
  1715000150000.logpoints.pb   <- zip #2
  ...                          <- sorted by key == sorted by time
```

That stream is the protobuf `LogPoints` message the recorder produced, re-chunked by the worker but unchanged in shape. The API unmarshals it and decrypts every point's `data` payload: the receiver encrypted each payload with the platform's symmetric AES key before publishing, and the worker, processor, and player API all decrypt with the same key. A point

that fails to decrypt is logged and kept as-is rather than allowed to sink the session — one corrupted event should cost you that event, not the replay.

Finally the API sorts, because the order events arrive in is not the order a human should see them. The rule: chronological by timestamp, and when timestamps tie, NAVIGATION events first, then MUTATION events, then everything else by sequence number. The reason is that a click and the DOM change it caused can carry the same millisecond, and the replay only makes sense if the page changes before the click that touched it is played. Chapter 16 shows how the player leans on this.

```
sort.Slice(items, func(i, j int) bool {
    if items[i].Timestamp != items[j].Timestamp {
        return items[i].Timestamp < items[j].Timestamp   // chronological
    }
    // Same instant: navigation first, then mutation, then sequence order.
    if items[i].Type == logpointproto.LogPoint_NAVIGATION &&
        items[j].Type != logpointproto.LogPoint_NAVIGATION {
        return true
    }
    if items[i].Type == logpointproto.LogPoint_MUTATION &&
        items[j].Type != logpointproto.LogPoint_MUTATION {
        return true
    }
    return items[i].Order < items[j].Order
})
```

The ordered result is marshaled back into a `LogPoints` message and written with the `Content-Type: application/x-protobuf` header. No JSON wrapper, no base64, no page of field names: the browser receives exactly the compact binary format it knows how to decode, in one round trip. This is the payoff of earlier design decisions. Events were captured as protobuf on the customer's page, encrypted before they touched a server, stored as compressed chunks, and are now decrypted and re-serialized to the same shape for the player — which is why a session of thousands of events still travels to the browser as hundreds of kilobytes rather than megabytes of JSON.

The heatmap and scroll endpoint family

The session endpoint serves one session; heatmaps only make sense across sessions, and nobody wants the API to decompress a hundred archives on every request to draw them. So the heatmap endpoints do not read archives at all. They read the derived collections the session processor wrote — `heatMapClicks` for individual clicks and `scrollHeat` for scroll attention — which were precomputed offline exactly so the interactive read path stays fast.

The family hangs under the same context path, all JSON, all behind the same JWT middleware:

```
GET /api/heatmap            ?companyId=&url=&deviceType=
GET /api/heatmap/aggregate  ?companyId=&url=&deviceType=&grid=
```

```
GET /api/heatmap/clicks     ?companyId=&url=&deviceType=
GET /api/heatmap/scroll     ?companyId=&url=&deviceType=
GET /api/heatmap/urls       ?companyId=
```

The base heatmap endpoint returns clicks grouped by element: one spot per element with a click count, keyed by the element's xpath, the stable cross-session identity — `lnrId` is a per-session counter that means nothing across sessions. The `/clicks` variant returns raw click rows, `/aggregate` buckets clicks into a grid of cells for coarser views, and `/urls` lists the distinct pages with click data so a picker can be populated. The scroll endpoint returns per-page depth bands with their reach (the share of sessions that scrolled at least that deep), dwell time, and share of attention, which the product draws as the scroll gauge you will meet in Chapter 16.

Three details make the family practical. First, the API normalizes the page URL by stripping its query string and fragment, so tracking parameters like `?utm_source=...` do not fragment one page into a thousand heatmaps. Second, every query is bounded by a configurable window — 90 days of click timestamps by default — and `deviceType` (mobile, tablet, or desktop, desktop by default) lets teams compare behavior per device class; an invalid device type earns a 400. Third, aggregated responses are cached in memory with a short time-to-live, about five minutes by default, because repeat views of the same URL should be instant. The aggregation queries rely on compound indexes the API creates in the background at startup — `heatMapClicks` on company, URL, and device type, plus a version field for `scrollHeat` — so a read never degrades into scanning a whole company's clicks. The version field matters quietly: the scroll reader filters for data written at version two or higher, the same process-version discipline the processor uses, so older records with a different shape are never served to a renderer that expects the new one.

The device chain

A session rarely exists alone. The same librarian or shopper generates many sessions across a day or a week, and support engineers asking "what happened next?" need to hop between them. The device chain endpoint answers with one indexed query: `GET /api/sessions/{id}/device-chain` returns every session recorded for the same company on the same device — identified by the device id the recorder sends with every batch — ordered by start time.

Each entry is a lightweight descriptor rather than a full session: session id, start time, the last URL visited, and the session's status. Status travels with the payload on purpose, because it is not safe to navigate everywhere. A session that is still ACTIVE or IDLE is still being written and its replay is incomplete; a REMOVED session has no archive left. The player therefore offers Previous and Next navigation only toward sessions whose status is FINISHED — complete, archived, and replayable. An index on company plus device id keeps the query fast even for devices that rack up hundreds of sessions.

What the player API deliberately does not do

The most important design decision here is negative space. The player API does not render anything: no DOM reconstruction, no cursor, no heatmap drawing. All of that happens in the browser, because rendering is a client problem and shipping pixels to every viewer would multiply cost for zero fidelity gain. The API's whole job is to authenticate, locate, decrypt, order, and serialize, then get out of the way.

By the same logic it does no analytics. The processor already derived errors, network analysis, and heat data offline; the API merely serves those collections. It does no re-derivation, no aggregation over raw archives on the hot path, and no long-lived connections. Its replicas are stateless — shared state lives in Mongo and object storage — so the ingress can scale it horizontally and a spike of people suddenly watching replays is absorbed by adding pods, not by tuning one server. The single write it performs, marking a session watched when opened, is explicitly best-effort and ignored on failure.

Failure modes, and sessions that are gone

Because the API sits at the tenant boundary, its errors are also a privacy boundary. Missing or malformed credentials earn a 401 with no data. An unknown session or user earns a 404. A valid token aimed at another company's session earns a 403. Bad parameters earn a 400, and storage failures earn a 500. In every case the response body is a short text message — never a partial payload, never a hint about what the data contained.

The most interesting case is the session that used to exist. The remover job deletes archived objects once a session passes the retention window (30 days by default, from Chapter 12) and marks it REMOVED. To the player API that session is indistinguishable from one that never finished archiving: the objects are gone, the download finds nothing, and the API returns an empty `LogPoints` payload with a 200. The player shows "no session data" and, when auto-advancing through a device chain, quietly skips to the next replayable session. This is deletion done right: when data is removed, the system cannot conjure it, does not fake it, and leaks no trace of what it once held.

> **Story checkpoint — Candlewood Books:** The library's portal is broken again, and Dana Whitfield is on the phone, polite but tired. Marta opens Dana's session from that morning — a narrow Safari window on a library iPad — and replays the last two minutes. The portal loads, Dana taps Add to Cart, and nothing visibly happens. Marta slows the playback and scrubs back: on that viewport the cookie banner sits squarely over the cart button, swallowing every tap. She screenshots the frame, sends it to the portal team, and a fix — a banner that yields on narrow screens — ships before Dana's afternoon shift ends. The contract stays.

Chapter takeaways

- The player API is a narrow read path: authenticate, authorize at the company level, fetch the archive from object storage, decrypt, order, and serve binary protobuf to the browser.
- Authorization is token-to-company, not token-to-session: the JWT's email claim resolves to a user who must hold an active membership in the session's company, or every endpoint answers 403.
- The cross-origin `lr_token` handoff moves the dashboard's JWT to the player origin, where it is validated, persisted, stripped from the URL, and re-attached as an `Authorization: Bearer` header.
- Archived sessions are reassembled from timestamp-named chunk zips under `sessions/{sessionId}/`, decrypted per payload with the shared AES key, and sorted with navigation and mutation events first on timestamp ties.
- Heatmap and scroll data come from processor-derived collections, bounded by a 90-day window, cached for seconds in memory, and served only to users who belong to the queried company.
- The device chain groups one device's sessions so the player can offer Previous and Next — but only toward FINISHED sessions.
- Gone is not broken: sessions that never archived, or whose retention window expired, return an empty payload, and the player treats "empty" as "no data," not as a failure.

Chapter 16: The Player: Reconstructing Time

Open a replay and you are watching a small miracle of engineering pretending to be trivial. A cursor glides across a page, pauses, clicks. Text appears in a search box. The page scrolls, a button is pressed, and four hundred milliseconds later two identical network requests leave the browser. It looks like a screen recording, but nothing on screen was ever filmed. Every pixel was rebuilt, from scratch, inside an invisible iframe, from a list of protobuf events that describe what happened rather than what it looked like.

That distinction — *describe* rather than *film* — is the subject of this chapter. The LogNroll player is an Angular application that decodes the session payload from Chapter 15, replays it on a clock, and reconstructs the page one event at a time. Understanding the mechanism also means understanding what replay can and cannot faithfully show — and that honesty is what makes the tool trustworthy.

From bytes to events

The journey starts where the player API left off: one `GET /api/get/{id}` request returns a binary `application/x-protobuf` body. The player requests it as an array buffer, so it arrives as raw bytes, and the first act of decoding is purely mechanical. The player ships the same protobuf contract as the rest of the platform — a generated `logpoint.ts` module from `LogPoint.proto` — and deserializing is one call that walks the wire format and produces an array of `LogPoint` objects, each carrying the six fields from Chapter 6: timestamp, type, version, order, index, and the opaque `data` bytes.

```
// Player decode path (simplified from the Angular service).
const buffer: ArrayBuffer = await http.get(url, { responseType: 'arraybuffer' });
const logPoints = LogPoints.deserialize(new Uint8Array(buffer)).items;
const decoder = new TextDecoder();                      // payloads are UTF-8
return logPoints.map(point => ({
  t: point.timestamp,                                   // epoch milliseconds
  p: LogPoint.LogType[point.type],                      // 'NAVIGATION', 'CLICK', ...
  version: point.version,
  index: point.index,
  d: decoder.decode(point.data),                        // JSON, HTML, or text
}));
```

The `data` payloads need their own decoding, because their contents vary by event type. Most are JSON strings describing the event in detail: a scroll's coordinates and target, a network request's URL and status, a form field's new value. A few are raw text: a full-page mutation carries the page's HTML snapshot, and a mouse-move payload is a compact, pipe-separated stream of coordinates and timestamps — `"412,318,1715000012345|421,319,1715000012390|..."` — which is how hundreds of cursor positions travel in a few

hundred bytes. Payloads are parsed only when a component needs them, so a long session never parses every JSON document up front.

The ordering rule, and why it matters

Decoding gives you a list of events, but a list is not a story. Events arrive out of narrative order for two reasons. The recorder batches events on the page and posts batches asynchronously, so the receiver, the worker, and the archive all see near-simultaneous events in slightly scrambled order. And events that share a timestamp have a required logical order: a click and the DOM mutation it triggered can carry the same millisecond, and replay only makes sense if the page changes *before* the cursor taps it.

The player API therefore sorts the payload before sending it, and the rule encodes a theory of how replay works. Events are ordered by timestamp, ascending. When timestamps tie, NAVIGATION events come first, then MUTATION events, then everything else by sequence number. Navigation and mutation are the events that change what the page *is*; clicks, scrolls, and key presses happen *on* the page. A page must exist before anyone can interact with it, so at any instant the structural events apply first.

```
sort:  by timestamp ascending
tie:   NAVIGATION first, MUTATION second, then by sequence (order)
why:   rebuild the DOM before replaying the interactions that touched it
```

The player trusts this ordering and leans on it. The first point defines the session's start time; the last — extended by the tail timestamps inside mouse-move batches and network response times — defines the end. Between those boundaries the player walks the list in order and applies each event exactly once. It keeps a set of already-applied points, so replaying forward never double-applies an event: the moment a point's timestamp passes the current playback time, the point is consumed — the DOM changes, the cursor moves, the request is logged. Because events apply incrementally, the page at any instant contains exactly the mutations whose timestamps have passed, which is what makes scrubbing feel like flying through time.

The playback clock

Replay needs a clock, and the player's is a simple loop. A state service holds the current timestamp and advances it on a timer: every 50 milliseconds while playing, the timestamp moves forward by 50 milliseconds times the playback speed. Available speeds are 0.5×, 1×, 2×, 4×, and 8×, and changing speed mid-play restarts the loop so the change applies immediately.

Each tick publishes the new timestamp, and components react through signals: the replay stage, the network panel, the console, the timeline, and the progress bar all observe the same

timestamp and update in concert. That shared clock is the entire trick of keeping panels synchronized with the page. When playback stops the timer clears; a scrub — on the progress bar, a timeline row, or an error — sets the timestamp directly and the player recomputes the page for that instant.

Seeking backward is where the design earns its keep. The player remembers which events it has applied, so when the timestamp moves backward it clears the applied set and rebuilds: mutations from the start of the session up to the new position re-apply in order, and cursor and scroll state are recomputed. Frame-by-frame debugging — the support engineer's most powerful habit — is exactly this: scrub back, press play slowly, and watch the page reassemble itself.

The transport controls sit in a toolbar under the stage: play and pause, a 10-second jump back and forward, the speed selector, and a progress bar that doubles as a scrubber. The bar also carries small milestone markers for events, so you can see where clicks and navigations cluster. Two toggles shape the experience: "Skip inactive," on by default, jumps the clock over dead time when the gap to the next meaningful event exceeds roughly a second; and the heatmap toggle, met later in this chapter, layers aggregate click and scroll data over the page.

Rebuilding the DOM

The heart of the player is a function that answers one question: given the next mutation event, how do I change the page I have already rebuilt? The page lives in an iframe whose document the player owns, and the player writes into it directly. Two kinds of mutation payloads exist, for two scales of change.

The first is the full snapshot. Some mutation events carry the entire rendered HTML — not the page's source code, but the DOM as it exists after the page's own JavaScript has run. The player loads it the way a browser loads any document: it opens the iframe's document, writes the HTML, and closes it. This is the replay's foundation; everything after it is repair work.

The second kind is incremental: a JSON document describing what a `MutationObserver` saw change — an attribute set, a text node edited, a child inserted or removed. The recorder stamps every element it touches with an attribute carrying its per-session id, so the player can find a mutation's target with a quick selector lookup, falling back to an xpath or element id when the stamp is missing. The player applies the three record types the observer produces. For **attributes**, the named attribute is set to its new value or removed when the new value is null, with the style attribute special-cased so cursor changes can be mirrored. For **character data**, a text node's content is replaced with the recorded text — how headings, prices, and error messages change on screen. For **child lists**, removed nodes are located (each removal is an id-and-xpath pair) and detached, while added nodes are materialized from their recorded structure: the player creates the element, stamps it,

recursively applies its attributes, styles, text, and value, and inserts it at the recorded position relative to its siblings.

That recursive materialization is what makes the reconstruction feel alive. When a React or Vue app swaps a list item, the recorder does not record “the framework did a thing”; it records the concrete nodes that appeared and vanished. The player creates real elements with real attributes and inline styles and appends them in the right order — a mutation payload is a small tree, and the player walks it the way a browser walks a document fragment.

```
{
  "type": "childList",
  "lnrId": "15270",
  "target": { "xpath": "/html/body/app-root/div[2]/ul", "tagName": "ul" },
  "addedNodes": [
    { "nodeType": 1, "tagName": "li", "lnrId": "15271",
      "attributes": { "class": "cart-item" },
      "children": [ { "nodeType": 3, "textContent": "Hardcover, 1" } ] }
  ],
  "removedNodes": [ "15269>>>/html/body/app-root/div[2]/ul/li[3]" ],
  "nextSibling": "15272>>>/html/body/app-root/div[2]/ul/li[2]"
}
```

Styles deserve their own note, because a page without its stylesheet is a page without layout. Styles arrive through STYLES events: either the text of a `<style>` element to append to the head, or a targeted update that replaces an element with a style block carrying the same id — the mechanism for dynamically injected component styles. The player also honors the recorded viewport: a META event near the start of a session records its width and height, platform, and user agent, and the player sizes the iframe to those dimensions. The recorded page might have been 1,440 by 900 pixels while the panel showing it is smaller, so a scale factor always fits the recorded viewport on screen, centered and letterboxed like a video player fitting a film to a screen.

One property of the reconstruction deserves emphasis, because it explains much of what replay can and cannot do: the page inside the iframe is inert. The iframe is sandboxed without script permission, and the player never executes the recorded page’s JavaScript. It does not need to — the snapshot captured the DOM *after* the scripts ran, and the mutation stream captured every change the scripts made afterward. Replay is the recorded consequence of code, not the code itself, and that is precisely why it is safe to watch. A malicious or broken script from the recorded site can never run inside the viewer’s browser — only inert HTML, CSS, and data cross the origin boundary.

A reconstruction, not a video

Every design decision above flows from one commitment: the player rebuilds what the browser showed by replaying the events that produced it, rather than replaying frames the

browser filmed. That commitment buys bandwidth, privacy, and searchability — a session is data, so it compresses, encrypts, queries, and stores for pennies — but it also draws the honest boundary of what replay can show: a high-fidelity *account* of a session, not a perfect copy, and knowing where fidelity breaks is part of using the tool well.

Canvas content is the first limit. When a page draws to a `<canvas>` element — a chart library, a drawing app, a WebGL scene — the recorder sees the element and its attributes but never the pixels inside it, because the browser does not expose them as DOM. A canvas-based chart appears as a blank or static rectangle; the interactions around it replay perfectly, but its contents are not part of the recording. Video is the same: a `<video>` element appears in the rebuilt page, but its playback timeline was not recorded, so you will not see the frame that was on screen. Cross-origin iframes are the third boundary: the recorder cannot see inside a cross-origin document — a payment iframe, an embedded map — so the iframe element replays, but its interior is whatever loads at replay time, or nothing. That is not a bug; it is the same-origin policy doing its job, and it is why a well-masked payment iframe is usually invisible to the recorder by design.

Animations are a subtler limit. The recorder captures state changes, not the intermediate frames of a CSS transition or animation. When an element animates from opacity zero to one over 300 milliseconds, the recording holds the before and after states; the replay applies the end state at the recorded instant, and the browser may or may not animate between them. In practice replay still reads clearly, because the eye forgives a missing tween far more readily than a missing element. What replay shows with total fidelity is structure and interaction — what was on the page, what the user did to it, in what order, at what time. What it shows approximately is motion and pixels.

Finally, the reconstructed page is a point-in-time artifact in a live world. Fonts and images referenced by URL load from the network at replay time, so a deleted hero image shows as an empty frame, and third-party content such as ads may differ from what the user saw. The recorded DOM is always faithful — that is the player's contract — but the resources it references are as fresh or as stale as the day you watch. The console, network, and error panels exist precisely to fill these gaps: when pixels cannot tell the whole story, the events that surrounded them can.

The cursor: mouse paths and click indicators

Every replay needs a ghost of the user: a small, absolutely positioned arrow drawn above the iframe, whose motion turns a trickle of batched coordinates into smooth movement.

Mouse movement is the volumetric enemy of session recording, so the recorder batches it hard: coordinates accumulate and flush at most every 200 milliseconds, each batch a single event whose payload is a pipe-separated string of `x,y,timestamp` triplets. When the player

reaches such an event during playback, it decodes the batch and walks the recorded positions, scaling each coordinate from the recorded viewport into the replay viewport by the ratio used for the page itself. A short CSS transition smooths movement between clock ticks, so the cursor glides rather than teleports. When the user scrubs or jumps, the cursor snaps to the last recorded position at or before the target time — during scrubbing you want the state of the world at the moment you land on it, not a re-enactment of the trip.

Clicks are where the ghost becomes emphatic. A click payload carries the viewport coordinates plus the element's identity — tag, per-session id, xpath, and the click's offset within the element. The player positions the cursor at the click point and triggers a brief ripple: the cursor swells with a yellow glow and settles, a visual echo of a finger meeting a screen. That ripple is among the most useful features in the product, because a support engineer scanning a replay does not read the timeline to find the moment of failure; they watch for the ripple that produced no visible result. In the Safari story from Chapter 5, the click that "did nothing" was exactly such a ripple, with the console panel beside the replay holding the error that explained why. The cursor even mirrors the recorded pointer style, becoming a pointer over a link and a text caret over an input — showing not just where the user pointed but how the page invited them to.

Keeping the scroll honest

Scroll is where naive replay breaks. A naive player would record the window's scroll offset and set it at replay time; that works until fonts load at different sizes or an image goes missing and the page ends up shorter than it was — so a fixed scroll offset now points at a different part of the page.

The recorder therefore captures scroll events with two views of the same fact. The absolute view records window or element scroll offsets and the maximum scrollable extent. The anchoring view records a `relativeTarget`: the id of an element near the top of the viewport when the scroll happened, with where that element sat relative to the viewport top and the scroll container's offset. At replay time the player prefers the anchor. It locates the recorded element in the rebuilt DOM — elements carry their recorded ids as attributes — and scrolls so that element lands at the same viewport position it occupied during the session. Content anchoring is robust to layout differences, because it asks "where is this element now?" instead of "how many pixels down was the page?"

```
{
  "x": 0, "y": 2100, "maxX": 0, "maxY": 6400,
  "target": { "element": "window" },
  "relativeTarget": { "lnrId": "9081", "top": 640, "scrollTop": 1460 }
}
```

Element scrolls — the scrollable lists, tables, and carousels inside a page — get the same treatment at smaller scale. The payload names the target by id, xpath, class, and tag, and records its offsets and dimensions. The player finds the element (by recorded id first, then xpath, then class plus tag) and applies the recorded scroll, scaled by the ratio between recorded and rebuilt element dimensions, because responsive layouts resize inner scrollers too. Whichever path fires, the player re-glues the heatmap canvas to the new scroll position, a detail that matters later.

Typing and input

What the user typed is the most sensitive data in a session, and it is handled with the discipline of Chapter 7: the recorder masks the fields it is told to mask, and masked values never leave the browser. Replay can only show what was captured, so a masked card field replays as a masked stand-in and a support engineer watching a checkout never sees a card number. Masking is not a replay feature bolted on afterward; it is a capture feature, and the replay simply inherits its honesty.

What the player does replay is recorded form state. Form events carry the element — its type, id, name, xpath, and per-session id — and the value it held at that moment. The player finds the element in the rebuilt page and applies the value by the control's kind: text inputs and textareas receive their text, checkboxes and radios their checked state, date and number inputs their typed values (set through the value and, where supported, the typed accessors so validation behaves), and selects have the matching options selected. The player then dispatches synthetic `input` and `change` events on the element, because the recorded page's own listeners are not running in the replay and the mutation stream usually captures their visible consequences anyway.

There is an honest gap worth naming: keystroke-by-keystroke typing is not animated. The recorder captures keyboard events, but the player applies a field's value at the moment the form event records it, so text appears as it was when the user finished typing. For debugging this is the right trade — you need the value in the field when the user pressed Pay, not a cinematic re-typing — though it is another reminder that replay shows the world at recorded instants.

Console, network, and the timeline panels

The stage shows what happened; the panels explain it. Around the replay sits a set of synchronized side panels, all watching the same playback clock, and their rows light up as the clock passes their timestamps.

The console panel replays the page's console. The recorder wrapped `console.log`, `warn`, and `error`, packing each call into a LOG event whose payload splits into the method and its arguments. The panel renders them as a scrolling console history, with navigations appearing as system lines ("Navigation to …"), and rows dim or highlight as playback passes them. This is the panel that made the Safari bug from Chapter 5 visible: the console error and the ripple that did nothing sat side by side — "works on my machine" becomes "here is the exact error, at the exact second."

The network panel is the other workhorse. Each NETWORK event carries the request and response — method, URL, headers, status, timing — so the player builds a table of every request the page made, filterable by type (XHR, CSS, JavaScript, fonts, other), status bucket (2xx, 4xx, 5xx, errors), and HTTP method; clicking a row opens the request details with headers and bodies when they were captured and not sanitized. Above the table, each request is drawn as a bar on a timeline strip: its left edge is the request's start time as a fraction of the session, its width the response time on the same scale. Two identical POSTs 400 milliseconds apart — the double-submit signature from Chapter 8 — show up as two bars nearly touching, a visual worth a thousand lines of server logs.

The errors panel merges two sources into one list. It parses console errors and failed network requests directly from the event stream — a LOG whose method is `error`, or a request whose stage failed or whose status was 4xx or 5xx — and it fetches the processor-detected behavioral errors for the session from the app API: rage clicks, dead clicks, and error clicks. Rows are tagged and color-coded, and clicking any row jumps the replay to that exact moment, cursor and all — turning a support ticket into a diagnosis in one click.

The timeline tab is the raw-material view for the curious engineer: every event as a row, filterable by type and searchable, with mutation targets summarized as readable paths; clicking a row seeks the replay to it. It is the panel you open to verify that a fix really removed the failing call, or to audit what a session contains.

The heatmap overlay and the scroll gauge

Heatmaps are where the player stops being a time machine and becomes a telescope across sessions. When the heatmap toggle is switched on, the player fetches the aggregated click data for the current page from the endpoints of Chapter 15 — element spots with counts for the overlay, scroll-depth bands for the gauge — and layers them over the replay.

The player classifies the replay's device type from the recorded metadata — user agent, platform, viewport width, mirroring the processor's classifier — and requests the matching data: mobile, tablet, or desktop. It normalizes the page URL exactly as the API does, stripping query and hash, because a heatmap belongs to a page, not its tracking parameters. To make the toggle feel instant, the player prefetches heatmap payloads for every URL the

session visited while the toggle is off, caching up to fifty page-and-device combinations; by the time you flip the switch, the data is usually already there.

Drawing the overlay is a study in canvas discipline. The heatmap canvas is sized to the *whole document*, not the viewport, and lives in the same scaled coordinate space as the page. Each aggregated spot is resolved against the rebuilt DOM: the player evaluates the spot's xpath, finds the live element, and places the heat at the recorded click offset within that element — the relative x and y the recorder captured — clamped to its bounds. Spots whose elements cannot be resolved, or that are currently hidden or covered — behind a modal, a carousel slide scrolled out of sight — are dropped rather than drawn at a guessed location: heat may only point at elements genuinely visible in the replay, so you see what was clickable, not a cloud of orphaned dots. A radial falloff builds an intensity field around each spot, weighted by click count, and the field is colorized through a blue-to-red ramp into the classic heat look.

Then comes the gotcha that shapes the whole overlay architecture. The canvas 2D API's `putImageData` call replaces the *entire* canvas — it does not composite over what is there, it wipes it. Once the heat field is committed, nothing else can be painted on that canvas, or it will be erased on the next wipe. The platform therefore keeps every interactive layer off the heat canvas entirely: the cursor, the hover highlight around an element, and the tooltip are DOM elements floating above the canvas, and the scroll gauge lives on its own separate canvas. A small rule — "after `putImageData`, draw nothing more on that canvas" — quietly determines the entire rendering architecture of the overlay.

```
ctx.putImageData(intensity, 0, 0);   // commits the heat field ...
// ... and wipes the WHOLE canvas — nothing may be drawn after it here.
// => cursor, highlight, and tooltip are DOM elements, not canvas paint;
//    the scroll gauge is a separate canvas.
```

The scroll gauge is the player's most distinctive piece: a tall, thin canvas mounted at the right edge of the replay, *outside* the replayed page, fixed in the player frame so it never scrolls with the content. It maps the page's whole scrollable range onto its own height, then draws one band per 10 percent of scroll depth, colored by the share of users' attention that band captured — the dwell share the processor computed with its viewport-visibility interpolation. A thin white line marks the maximum depth anyone reached, and hovering a band shows the numbers behind it: how far down it is, what share of users reached it, how many seconds it spent in view, what share of attention it earned. The gauge turns "nobody scrolled past the fold" from an intuition into a measurement.

Above all, the gauge carries a live **"you are here" marker**: a small glowing line that tracks the replay's own scroll position as the session plays. The player binds scroll listeners inside the replay iframe — on the window and, in the capture phase, on the document, so inner element scrolls are caught too — and every scroll schedules a redraw through `requestAnimationFrame`, coalescing a busy page's many scroll events into one paint per frame.

As the recorded user scrolls, the marker slides down the gauge, and you see one person's depth against the crowd's in real time.

Watching like a television

The replay stage is built to be watched, not clicked — the codebase literally names the component after a television. The stage always fits the entire recorded viewport on screen, letterboxed, so nothing important hides off-screen and the layout never jumps mid-session. Panels can collapse until only the page remains, and the toolbar's auto-play mode turns the player into a channel: when playback reaches a session's end, the player advances to the next FINISHED session in the device chain and starts it, continuing until the queue runs out or the viewer stops it. A support lead reviewing a day's problem sessions back to back can set the player loose and watch the queue drain.

The device chain deserves a moment, because it answers the question replay cannot ask alone: "what happened after?" The player fetches the device's session list from the API, finds the current session's position, and enables Previous and Next — but only toward FINISHED sessions. An ACTIVE or IDLE session is still being recorded, and its replay would end mid-story; a REMOVED session has no archive. So the chain hops between finished sessions, and empty ones are skipped automatically during auto-play, with a safety cap on consecutive skips so a corrupted stretch cannot send the player into an infinite loop. For support the chain is a superpower: Dana's frustrating afternoon is not one session but six, and the engineer who opens session three can step to four with one click and watch the problem compound.

Performance under the hood

A replay is a streaming workload disguised as an interactive one, and the player leans on a handful of techniques to keep long sessions fluid. The playback clock runs outside Angular's change-detection zone — advancing the timestamp deliberately avoids triggering Angular's machinery on every tick, because running change detection twenty times a second across the component tree would make a replay stutter. Components observe the timestamp through signals and update only what they must.

Rendering work is coalesced and bounded. Heatmap and gauge redraws funnel through `requestAnimationFrame`, so however many scroll events or mutations fire between frames, each canvas paints at most once per frame. Scroll-triggered heatmap re-checks are debounced — a full document-space redraw on every scroll frame would be too expensive on long pages, so the player waits for scrolling to settle, then re-evaluates which spots are visible and hit-testable. The document-sized canvas is capped at 8,192 pixels per side, a pragmatic

ceiling above which a page is too tall to draw usefully. Events apply incrementally rather than re-rendering each tick: an event is consumed once, and later ticks only notice newly due events. Even the panels cooperate — rows flip to "played" styling as the clock passes them, a class change rather than a rebuild.

The one honest cost is the payload itself. The player fetches the full assembled session in one request, so a very long or event-dense session arrives as a large binary blob and lives in memory for the duration of the viewing. The chunked archive layout from Chapter 11 exists partly so this could one day be fetched progressively — the pieces are timestamp-named and independently retrievable — and Chapter 19 does the arithmetic on how far a single payload stretches. For the sessions most people debug — a few minutes, a few thousand events — the current approach is instant, and skip-inactive and scrubbing make even long ones bearable.

> **Story checkpoint — Candlewood Books:** Priya opens the double-charge session and presses play. The customer adds a book, taps Pay — nothing. Taps Pay again, harder. Priya pauses, scrubs back ten seconds, and slows the replay to half speed. Two ripples land on the button; beside the page, the network panel shows two identical POSTs to the payment endpoint, 400 milliseconds apart, both answered 200. She clicks the first, then the second: same body, same amount, the same pattern the gateway logs had hinted at. The fix — disable the button on submit, add an idempotency key — ships that afternoon, and the "charged me twice" tickets stop arriving the following week.

Chapter takeaways

- Replay is reconstruction, not video: the player decodes protobuf events and rebuilds an inert HTML page in an iframe by applying recorded mutations in order; no page script ever runs.
- The payload is ordered by timestamp with NAVIGATION and MUTATION events forced first on ties, so the page exists before the interactions that touched it are replayed.
- A 50-millisecond playback clock at 0.5× to 8× speeds drives everything; scrubbing backward clears the applied-event set and rebuilds the page up to the new position.
- Mouse paths replay from batched `x,y,time` payloads with a click ripple on every click; scroll replay anchors to recorded elements rather than raw pixel offsets, so layout drift cannot break it.
- Form values replay per control type with synthetic events, but only what was captured — masked fields replay masked, and typing is not animated.
- Console, network, error, and timeline panels all watch the playback clock; the network panel's request bars make a double-submit visible as two POSTs 400 milliseconds apart.

- The heatmap overlay resolves every click spot against the live rebuilt DOM and drops spots for hidden elements; because `putImageData` wipes the whole canvas, the cursor, highlight, and tooltip are DOM layers, and the scroll gauge is a separate fixed canvas with a live “you are here” marker.
- Playback ticks run outside Angular change detection, canvas redraws are rAF-coalesced and debounced, and heatmap data is prefetched per page so toggles are instant.

Chapter 17: The Product Around the Replay

For the last several chapters we have followed a single session through the machinery: captured on the page, shipped to the receiver, parked on the message bus, filed into the archive, analyzed by the processor, and finally served back to the player. That is the plumbing. But nobody logs into a session replay platform to admire the plumbing. They log in to answer a question — "why did the checkout fail for this customer?" or "is anyone actually reading the new landing page?" — and the answer usually starts in a list of sessions, not in a single replay.

This chapter steps out of the pipeline and into the product that wraps it: the dashboard where sessions become a searchable, filterable, ranked surface. The reference implementation's admin application is an Angular dashboard organized around a single company workspace, and nearly every capability we have discussed as raw machinery — the session document, the error collection, the heatmap clicks, the scroll-heat buckets, even the uptime probes — reappears here as a tab, a table, a chart, or an email.

The session list is the front door

Open a company in LogNroll and the default landing surface is the session list. The company workspace renders a set of lazy-loaded tabs — CRM, Sessions, MCP, Errors, Monitoring, Discovery, Team, Plans, Settings, and Integration — that are defined once in a single source of truth and reused by both the in-page tab strip and the side menu, so the navigation can never drift between the two. Sessions sits at the heart of it.

The list itself is a dense table. Instead of dozens of narrow columns, the dashboard groups related facts into composite cells: a User cell (name, email, or "Anonymous" when the visitor never identified), a Summary cell, Activity and Date, a Context cell that bundles location and platform, then URL, Device, and Session ID. A counter above the table reports the total number of sessions matching the current filters, and paging is server-side, because no browser should ever hold ten thousand session rows.

Two filtering mechanisms work together. A search/filter picker groups its options the way an investigator thinks: by **User** (User Name, User ID, User Email, Identified), by **Session** (Session ID, URL, Start Time, Processed, Duration), and by **Device and context** (Device ID, Device type, Platform, OS, Location, Browser). Typing in the user search hits an autocomplete endpoint that suggests names and emails harvested from real sessions. Quick filters sit alongside for the common questions: Today, the last 7 or 30 days, Identified versus Anonymous, Processed, Long sessions (the product defines "long" as five minutes or more), and device classes — Mobile, Tablet, Desktop, iPhone, Android. Each preset is a pre-baked

query over the same session data the pipeline has been maintaining all along: location comes from the IP geolocation the receiver performed at ingest, the device and platform from user-agent parsing, the user from an IDENTIFY event the recorder captured, and the URL from the NAVIGATION stream.

The elegant part is how little of this is bespoke. The list is a read over the `sessions` collection we met in the storage chapter — the same documents the receiver creates, the status job finishes, and the worker archives. Every column is metadata the hot path already collected for its own reasons, now indexed and served as a product. And each row's play button deep-links into the player on a separate origin, passing the session id and a short-lived token, so the jump from "found the session" to "watching the session" is one click.

Errors: from console to digest

Replay answers "what happened," but errors answer "what went wrong, and where does it happen most." The Errors tab aggregates the processor's `sessionErrors` output — console errors, uncaught exceptions, and behavioral signals the dashboard describes honestly as rage, dead, and error clicks — into groups a human can triage. Drop-downs filter by period and by error type, and an ignore-filter mechanism lets a team mute known noise so a recurring third-party warning does not drown out a real regression.

The same aggregation powers the daily error digest. On a schedule (default nine in the morning), a job in the main application pulls each company's aggregated errors and emails them through the mail-rendering service — a plain-language summary of what broke across your users' sessions yesterday, with the counts that tell you whether it is a blip or a spike. The Settings tab hosts the alert toggles for the daily digest and for host-monitoring alerts, each with a test button that sends a sample message to the authenticated user. This is the product's attention-routing layer: no support team can watch every session, but an email that says "seventeen users hit the same checkout error yesterday" tells them precisely which replays to open. The processor chapter showed how that loop caught Candlewood's Safari bug the same day; here we see the surface that delivered the news.

Heatmaps and scroll heat: the aggregate view

Where the error digest answers "what is breaking," the heatmap answers "where are people looking and clicking." The overlay itself lives in the player — you watch a session and see click density painted over the reconstructed page, with a live scroll gauge at the edge showing how far users reached. But the same processor output supports aggregate endpoints served by the player API: click heatmaps, scroll-heat distributions, and the URL lists they hang off.

The unit of aggregation is deliberately the page element, keyed by xpath rather than by the per-session element ids the recorder assigns. Because xpath is stable across sessions, clicks from a thousand visitors on the same checkout button accumulate into one meaningful hot spot; scroll-heat processing bins dwell time into ten percent bands of the page with viewport-visibility interpolation, so the product can report both reach (how far down people actually scrolled) and attention (how long they lingered per band). For a product team, this turns replay from a microscope into a map: replay shows one user's journey frame by frame, while the heatmap shows the shape of every journey at once.

The analytics surface beyond sessions

Around the session-centric core, the dashboard carries surfaces that treat users and URLs, not sessions, as the primary object. The CRM tab organizes sessions by the people behind them — identified through the recorder's `identify` calls — into a user-centric board with contact profiles and analytics, so a support lead can pull up "everything this customer did lately" instead of fishing through raw sessions. The Discovery tab mines session URLs for recurring navigation patterns, surfacing candidate flows — checkout sequences, onboarding paths — as structured patterns rather than leaving them implicit in the data. There is even an MCP tab that walks a user through connecting an AI assistant to the company's session data, which we will meet properly in the future-facing chapter.

These tabs are not separate products bolted together. They all read the same derived collections — sessions, errors, heatmap clicks, scroll heat, and the URL streams — and they all carry the same company scoping. That shared substrate is what makes the suite feel like one system rather than five tools.

One platform, many companies

Everything above exists inside a tenant boundary, because LogNroll is a multi-tenant service: one deployment, many customer companies, strict separation between them. The data model is three collections. A **company** is the top-level workspace. A **user** is a person with an account, identified by email and authenticated with magic-link codes. Between them sits the membership document in the `team_members` collection, which binds a user to a company with one of three roles — **OWNER**, **ADMIN**, or **MEMBER** — and a status of ACTIVE or REMOVED.

Roles gate what a person can do. Only an owner or admin manages the team: members cannot remove other members, and owners cannot be removed at all. Invitations arrive by email and create the membership when accepted. Onboarding walks a new company through the steps — verifying the email address, creating the workspace, and reaching the integration

instructions that show exactly where the snippet goes. The Integration tab keeps those instructions and a check that confirms recording is actually arriving.

The consequence for the product surface is that every tab is company-scoped by construction. The session list, the error groups, the monitoring hosts, the plans — each request is validated against the caller's active membership, which we will examine as a security property in the next chapter. From the product's point of view, the important thing is simpler: ten companies using the service feel like ten separate products, because each user genuinely only ever sees their own.

Plans and billing

A commercial replay service must turn sessions into revenue, and the billing surface is deliberately boring. Each company has an active plan, drawn from a catalog of plan types — free, regular, contact-sales, custom, pay-as-you-go, and unlimited — billed monthly or yearly. The plan service tracks usage against the plan, and invoices accumulate as records. Payment runs through a dedicated gateway microservice integrated with a Ukrainian payment provider, with signed requests and callback retries, so a dropped webhook does not silently lose a payment. The free tier is what gets a small shop like Candlewood Books in the door; the plan page is where they choose a retention window and limits that fit.

None of this is architecturally exotic. What matters is that billing sits in the same control plane as the rest of the company data — the same application, the same team-membership checks, the same company id in every URL — rather than dangling off a third-party widget.

Replay plus everything else

Watch how an investigation actually flows through this surface, and the composition becomes clear. The morning digest email arrives: seventeen users hit the same checkout error. In the Errors tab you filter to that group and open one affected session in the player. The console panel shows the exception; the network panel shows the request that preceded it; the replay shows the exact click. You fix the bug. Later, in the heatmap for the same page, you notice the payment button is barely reached on mobile because users stop scrolling above it — a UX finding no single replay would have made obvious. Meanwhile the Monitoring tab reports the origin stayed up all night, so you know the outage you half-expected never happened.

That is the product thesis of a session replay suite: replay, errors, heatmaps, analytics, and monitoring are one workflow, not a feature checklist. Each surface answers the question the others cannot — replay supplies the ground truth of what a user did, errors rank what broke, heatmaps reveal what attracted attention, and monitoring confirms the infrastructure

behaved. They compose because they share a session id, a company boundary, and a set of derived collections.

> **Field note:** Replay alone is a lens, not a product. A single reconstructed session is compelling for exactly one investigation at a time, and no team can watch ten thousand of them. The dashboard is the curation layer that makes ten thousand sessions usable — it filters, ranks, aggregates, and summarizes until the handful of sessions worth watching rise to the top. The replay is the payoff; everything around it exists to find the right replay in the first place.

Chapter takeaways

- The dashboard is a curation layer over the same collections the pipeline already maintains: the session list, error groups, and heatmaps are queries, not new systems.
- The session list filters by what the hot path already captured — user identity from IDENTIFY, device from user-agent parsing, location from IP geolocation, URL from navigation — with quick presets for time windows, identification state, and device class.
- The error digest (a scheduled aggregation emailed daily) and the Errors tab route attention from “a bug happened” to “the replay worth watching.”
- Heatmaps and scroll heat aggregate across sessions by stable xpath identity, turning single-session replay into page-level maps of clicks and attention.
- Multi-tenancy is a three-collection model — companies, users, and team memberships with OWNER, ADMIN, and MEMBER roles — and every product surface is company-scoped by construction.
- Plans are typed and data-driven (free, regular, contact-sales, custom, pay-as-you-go, unlimited), billed monthly or yearly through a dedicated payment microservice with callback retries.
- Replay, errors, heatmaps, analytics, and monitoring compose into one investigation workflow because they share one session id, one company boundary, and one set of derived data.

Chapter 18: Operating a Session Replay Service

By now the architecture should feel familiar: a recorder in the browser, a thin receiver, a durable bus, a worker that files sessions into cold storage, a processor that derives insight, and a player that serves it back. Between the diagram and the reliable service sits the operational work — keeping one customer's data invisible to another, absorbing crawlers and hostile payloads, scaling each tier independently, surviving Kubernetes restarts and deploys, honoring retention promises, and giving support teams a workflow they can trust. This chapter is about that work, using LogNroll as the running example: a deliberately small platform that still has to behave like a serious one.

Multi-tenancy and isolation

LogNroll runs one deployment for many customer companies, so isolation is not a feature; it is the invariant everything else must not violate. The design leans on two reinforcing layers: every read is scoped to a company, and every read is re-authorized against live membership rather than trusting a token alone.

The company boundary in the data

Each session document in the shared `sessions` collection carries the `companyId` of the company that recorded it, and every product surface is organized around that id — error groups, monitoring hosts, and billing under `/companies/{companyId}/...` routes, and the session list as a company-filtered query over `sessions`. The discipline is enforced in the application layer: the session component validates company access on every request — listing sessions, opening one, reading its errors, updating its metadata — by checking that the calling user has an ACTIVE membership for that company in the `team_members` collection. A user with no such membership is denied before any query runs, and the denial is logged as a suspicious-access event.

Membership is not static, and the status model is what makes revocation real. When a person leaves a company, their `team_members` document flips to REMOVED, and the next request simply fails the active-membership check. There is no cache to wait out and no token to blacklist; the authorization decision is a database lookup on every call. Roles sharpen the same model: only an OWNER or ADMIN can manage the team, a MEMBER cannot remove another member, and an OWNER cannot be removed at all, so a company always retains someone who can administer it.

Tokens that prove identity, not access

Authentication across the control plane and the player uses JWTs signed with a shared HS256 secret and issued by the main application after a magic-link email code. The dashboard and the player live on different origins and cannot share cookies, so the dashboard passes its token through a short-lived `lr_token` query parameter on the deep link; the player frontend consumes it and strips it from the URL immediately.

The subtle operational point is that the token proves who you are, not what you may see: a session id alone must never be a capability. A leaked replay URL without a valid, still-authorized token is just a 403.

Shared storage, enforced boundaries

The archive is shared infrastructure. The worker uploads each session to one of two object-storage backends — Spaces buckets in two different regions — selected per session through the session's storage service name. These are storage tenants of the platform itself, not per-customer buckets; sessions from every company share the same buckets, addressed by session id under the `sessions/{sessionId}/` prefix. Isolation therefore comes not from where the bytes sit but from the authorization layer in front of them: before the player API decrypts and serves a single byte of archive, it confirms the caller is an active member of the session's company.

That arrangement inverts a common intuition: partitioning storage per customer is a reasonable future step, but it does not remove the need for the membership check on the read path. The storage topology may change; the authorization boundary may not.

Quotas, crawlers, and abuse

A replay receiver is a public endpoint that accepts arbitrary POSTs from arbitrary browsers, which makes it a magnet for everything that crawls or scans the internet. The ingestion path defends in three layers.

Crawlers are rejected, not just asked to leave

`robots.txt` is a request, not a rule, and compliance is voluntary. The receiver therefore serves a robots file that disallows every crawler — a wildcard group plus explicit named groups, because several AI crawlers only honor a group that names them — and stamps every response with `X-Robots-Tag: noindex` so nothing on the host gets indexed. Beyond that, a middleware layer rejects known AI crawlers and content scrapers with a 403 before they reach any handler, matching a curated list of LLM-training and scraping user agents. The

blocked requests still pass through the request-logging middleware, so the rejections appear in the access log rather than vanishing silently.

One nuance is deliberate: search-engine bots such as Googlebot are not hard-blocked, even though robots.txt fully disallows them. Returning 403 to a search engine can trigger "soft-404" behavior and other indexing side effects, so the polite disallow is trusted for them while the aggressive reject is reserved for crawlers that ignore politeness.

Bad payloads and hostile requests

A telemetry endpoint must assume some requests are junk. The ingress layer caps request bodies at 20 MB before they reach the service. The receiver validates what arrives: an unparseable batch is answered with a 400 rather than an exception, and an unknown company or session yields a 404. The hot path is deliberately thin — the receiver does not decode archives, run analytics, or touch object storage — so the surface an attacker can reach stays small.

Timeouts are treated as a correctness issue because of the browser. The receiver wraps its ingestion handler in a 20-second timeout, and the middleware stack ensures every response carries CORS headers. If a request stalls, the receiver answers with a readable 503 instead of letting the CDN's longer origin timeout produce a gateway error page — which would carry no CORS headers and surface in the browser as an opaque network failure the logger could neither interpret nor retry.

Sessions as a resource

The subtler abuse is volumetric rather than malicious: every visitor to a customer's site opens a session, and a popular site produces many of them. The platform answers with a cascade of natural backpressures rather than a single quota gate: the recorder batches events client-side, the status job moves stale sessions toward FINISHED, the worker purges each session's NATS subject after archiving so the bus never accumulates dead sessions, and the remover deletes archived data after the retention window.

Scaling the tiers

A replay pipeline is not one service to scale but a sequence of tiers with different scaling personalities. The lesson generalizes: the stateless layers scale by adding replicas; the stateful layers are sized and protected instead.

Receivers are stateless by construction. A receiver holds no session state, no local archive, and no queue; the only shared state it touches is the MongoDB session document and the NATS subject it publishes to. Any receiver can therefore handle any batch, which makes the

tier trivially horizontally scalable behind the ingress, with pod autoscaling configured on the deployment. When a customer's traffic spikes — Candlewood's promo weekend in the story, a flash sale for someone else — the answer is more receiver pods, and the bus absorbs the burst between the stateless front and the bounded workers behind it.

The message bus is the shock absorber, and sizing it is mostly a disk and retention question. JetStream persists messages to disk so a worker restart loses nothing; one subject per session preserves per-session ordering without any coordination protocol; and because the worker purges each subject once the archive is written, the stream's disk footprint tracks the ingestion backlog rather than cumulative volume. Delivery is at-least-once, and the downstream claim pattern makes duplicates harmless.

Workers scale by pool, but pool members must not race. A worker claims a FINISHED session with an atomic find-and-update on lock fields, writing its instance id and a lock timestamp that expires after roughly two minutes. A crashed worker's claim goes stale and another worker can reclaim the session; a live worker never double-processes because the claim is atomic. Scaling the pool means adding claimers, not duplicating work.

The true coordination point, and therefore the hot spot, is the shared `sessions` collection in MongoDB. Every tier reads or writes it — the receiver updates session metadata on ingest, the status job transitions states, the worker, processor, and remover all claim through it, and the dashboard queries it. Several mitigations keep it calm. The receiver throttles its metadata writes instead of updating on every batch. Claims are single atomic operations rather than read-then-write races, and the collection is indexed on the fields the hot queries filter by — `companyId` for every dashboard query, plus the status and lock fields the claim queries use. The derived collections (errors, heatmap clicks, scroll heat, backend requests) are written by the processor after archiving, off this store, so the coordination collection never grows into a general-purpose analytics database.

Operating on Kubernetes

Every deployable service ships as a container with a Helm chart, and the operational conventions repeat across all of them.

Probes: the two-port actuator pattern

Kubernetes needs to know whether a container is alive and ready, and a health check that shares a port with real traffic is a bad idea — a busy request handler can make a naive health endpoint slow, and health traffic can contend with the traffic it protects. The convention here is the **two-port actuator pattern**: the application serves on its traffic port (default 8080), while a second server — Spring Boot's actuator in the Java services, an explicitly started

actuator server in a goroutine in the Go services — listens on a separate actuator port (default 8181) and answers only health routes:

```
traffic port 8080    app routes only
actuator port 8181   /actuator/health
                     /actuator/health/liveness
                     /actuator/health/readiness
```

The Helm charts point the liveness and readiness probes at the actuator port with short initial-delay settings, and the Go services start the actuator in its own goroutine so a probe can never block the request path. Even the MCP server follows the same contract, running its actuator unconditionally so the probes pass in any mode.

Cron jobs versus long-running services

LogNroll runs two kinds of scheduled work, and the distinction is deliberate. The session-pipeline jobs are Kubernetes CronJobs: the status job runs every minute to advance session lifecycles, and the remover runs hourly to delete sessions older than the retention window. CronJobs fit because the work is bounded, idempotent (the lock pattern makes reruns safe), and not always-on; if a run fails, the next scheduled run retries the same claim-based work.

The control-plane jobs inside the main application are different: they are in-process schedulers inside a long-running service, because they are small, frequent, or tied to application state. The uptime checker ticks every 60 seconds and probes each monitored host on its own cadence; the certificate check runs daily; host discovery scans session URLs each night (03:30 by default); and the error digest fires every morning at nine. Running these inside the service keeps their configuration and secrets alongside the code that uses them.

Secrets and observability

Configuration and credentials come from a single shared Kubernetes secret named `lognroll`, referenced by name in each chart and mounted into the deployment as environment variables — the MongoDB URI, the AES encryption key, the NATS credentials, mail and payment keys. Nothing secret lives in a Helm values file or an image. CI keeps its own credentials as repository variables, decoded only inside the pipeline step.

Observability is metric, log, and probe. The Go services expose Prometheus metrics (the worker's metrics package is the fullest example), and the Java services expose Spring Boot Actuator endpoints with Prometheus support. Logs are structured JSON, and the receiver logs a per-boot marker including the pod hostname at startup — a deliberately boring line that becomes invaluable during incident correlation, because a gap in the access log bounded by two boot markers is the signature of a restart, an out-of-memory kill, or a rolling deploy.

The deploy story

Deployments flow through Bitbucket Pipelines using a shared library repository whose reusable steps the sibling services import. A pipeline for a Java service runs a Maven package step on a JDK 21 image, then a Docker build and push with an environment-derived image tag — date plus build number, with a release suffix for main-branch builds — then a Helm upgrade into the target cluster. The deploy step runs an install-or-upgrade with a wait and timeout against the chart's `helm/dev` or `helm/prod` values and the freshly built image tag. Dev and production are separate namespaces on separate clusters, and every environment carries its suffix through the whole stack: `-dev` versus prod hosts, NATS subjects such as `sessions3-dev`, and database names taken from the MongoDB URI path.

The pipelines also make one operational reality visible: the platform is mid-migration. The Go rewrites of the receiver, worker, and player API run in dev while production still runs the Java originals, and two Java jobs were split into Go cron jobs. Deploying safely therefore means knowing which stack each environment expects, and the ingress values encode that knowledge host by host. It is a temporary operational tax — the price of migrating a working system incrementally.

GDPR and retention operations

For an EU-built product that records real users' screens, privacy is operational before it is legal: retention must be a mechanism, not a policy. The recorder masks sensitive inputs at capture time, so card numbers and passwords never reach the archive, and consent is a customer-site concern the SDK supports. The platform's own copy of the data is bounded by design: the status job closes dead sessions, the worker purges NATS subjects after archiving, and the remover deletes archived sessions older than the retention window — 30 days by default — by claiming each FINISHED session, deleting its archive objects, and marking it REMOVED in the coordination store. A REMOVED session is gone from the player's perspective: the player API returns nothing for it, and the archive no longer exists to be read.

Retention is configured per plan, which makes it a product decision with operational teeth — a customer on a longer window simply changes the cutoff that the same machinery enforces. Honest deletion must also reach every store a session passed through: the coordination document, the archived objects, any still-queued NATS messages, and the derived collections the processor wrote. LogNroll's lifecycle automates the archive and coordination sides; a full account-level deletion is the kind of flow an operator walks through deliberately, store by store, rather than assuming one flag cleans everything. This book makes no certification claims for LogNroll; the discipline described here is the groundwork those certifications would later audit.

The support workflow, rebuilt on replays

The operational payoff of the whole platform is most visible in support. A customer calls with a problem — an order they cannot place, a button that does nothing. The support agent asks for consent to look at the session, opens the dashboard, filters the session list by the customer's email, and watches the replay. The console panel shows the error the browser reported; the network panel shows the request that failed; the replay shows the click that triggered it. The agent sees the bug, the engineer fixes it, and the customer gets one email: here is what happened, here is what we changed.

Three properties make that workflow safe at scale. First, access is scoped: the agent only ever sees sessions of their own company, and the short-lived replay handoff dies in the URL bar. Second, redaction happened at capture, so the replay the agent watches already excludes the masked fields — the support conversation never involves handling raw card data. Third, when another person needs to see a session, the answer is membership, not a forwarded link: an engineer joins the company team, and revocation is instant when the membership ends. Consent is the human half of the same discipline — the customer agreed, the agent respects what the customer shared, and the platform makes it technically impossible to over-share.

None of this requires the support agent to understand the architecture — which is the point. The isolation, quotas, scaling, probes, and retention described in this chapter exist so that a small bookstore's support agent can confidently tell a customer, "I saw what happened, and it is fixed."

Watching the watcher

A session replay platform should be its own best customer, and LogNroll closes the loop in a few honest ways. Every service exposes health endpoints under the two-port contract, and the receiver adds a heartbeat path, so the cluster's probes and any external checker can watch the platform with the same tooling its customers use.

The most fitting dogfood is the host-monitoring feature itself. Its discovery job learns origins from the URLs in recorded sessions, normalizing away localhost, loopback, private, and link-local addresses — which means the public origins LogNroll itself operates, such as the dashboard host, are exactly the kind of origin the feature can watch. The same uptime probes and TLS certificate checks that alert a customer that their storefront is down can watch LogNroll's own public services, and the development demo site exercises the recorder against the dev pipeline so the team sees real sessions of its own product. The platform that replays other people's users watches itself with the same lenses: replays when a bug needs a witness, errors when something breaks, heatmaps when the product team wonders where people click, and uptime checks when the question is simply "is it up."

> **Story checkpoint — Candlewood Books:** Marta's support routine now opens with the morning error digest and a quick scan of flagged sessions, and it scales past her: when a seasonal hire joins for the summer rush, Marta does not hand her a script. She walks the new agent through three real replays of real incidents — names masked, access scoped to the company dashboard — showing how to read a timeline, spot a rage click, and write the one-email answer. Sharing stays inside the dashboard, where access follows team membership and disappears the day a member leaves, so the seasonal hire can see sessions but never export or forward them. By August the new agent closes her own tickets in an afternoon; Marta spends the extra hours on the digest instead of the queue, and her reply time — from a day of back-and-forth to one afternoon — has become the standard the whole team is measured by.

Chapter takeaways

- Multi-tenancy is enforced at authorization, not by storage topology: every session read is validated against the caller's ACTIVE company membership, and the player API re-checks membership before serving archive bytes.
- JWTs prove identity only; access is re-derived from live `team_members` state on every request, so revoking a member takes effect immediately.
- Crawlers are handled in depth: a disallow-all robots file, `X-Robots-Tag: noindex` on every response, and 403s for known AI crawlers — while search engines are politely disallowed rather than hard-blocked.
- The pipeline scales by personality: stateless receivers add replicas, JetStream absorbs bursts with per-session subjects and post-archive purges, workers claim sessions through atomic two-minute locks, and the shared `sessions` collection is protected by write throttling, targeted indexes, and atomic claims.
- Kubernetes conventions are uniform: the two-port actuator pattern keeps health traffic off the app port, pipeline jobs run as CronJobs while small frequent control-plane jobs run in-process, and all secrets come from one shared cluster secret.
- Deploys flow through a shared Bitbucket Pipelines library: Maven and Docker builds with environment-scoped image tags, then Helm install-or-upgrade into separate dev and prod namespaces with environment suffixes throughout the stack.
- Retention is an operational mechanism: the remover deletes archived sessions after a configurable window and marks them REMOVED, and honest deletion must reach every store a session passed through.
- Support workflows are safe at scale because access is company-scoped, redaction happened at capture, and sharing runs through revocable membership rather than forwarded links.

Chapter 19: Performance Math and Cost Engineering

Every chapter so far has described what a session replay platform does. This one is about what it pays. A session replay service is an accounting problem wearing an architecture: every captured interaction is a few bytes that must be encoded, shipped, buffered, stored, and later read back, and each step has a price in CPU, bandwidth, disk, and money. The numbers decide the design. The recorder batches because unbounded requests would be too chatty; the receiver throttles its database writes because per-event writes would melt the database; the worker archives in chunks because thousands of tiny objects would cost more than the bytes inside them. None of that structure is aesthetic. It is arithmetic. This chapter does the arithmetic in the open — real constants where LogNroll's code fixes them, clearly labeled rough estimates elsewhere. Work the examples with your own assumptions; the method matters more than the answers.

The cadence of a captured minute

The first number to pin down is how many events one user generates, and it starts in the recorder, whose constants are real and drive everything downstream. The recorder never sends one event per user action: the queue flushes on a **100 ms** interval (`batchDelay`), each batch holding at most **200** events (`batchSize`), and high-frequency captures batch again before that. Mouse movement is collected for up to **200 ms** before becoming an event, keyboard activity waits **100 ms**, and form changes are debounced **500 ms**. A mouse point is recorded only if the pointer moved at least **one pixel** since the last recorded point. These valves keep a talkative browser cheap.

Now do the arithmetic of an engaged minute. While the pointer is actually gliding, a mouse event forms roughly every 200 ms, so a fully continuous minute of movement yields about 300 mouse events, each carrying a pipe-delimited string of the coordinates sampled in its window. Real interaction comes in bursts of motion between reading, so model an active minute as a mix:

Per user, one active minute (illustrative)	Low activity	Typical	Heavy
Mouse move events (batched at 200 ms)	5–15	30–80	150–300
Scroll events	2–5	5–15	20–40
Clicks	0–2	2–5	5–10

Per user, one active minute (illustrative)	Low activity	Typical	Heavy
Keyboard events (batched at 100 ms)	0	5–20	30–60
Input / form events (debounced 500 ms)	0	1–3	5–10
Network events (XHR/ fetch wrapper)	1–3	5–15	10–30
Console log events (polled each second)	0–2	2–10	10–40
DOM mutations and everything else	5–20	20–100	100–500
Total events	**15–50**	**70–250**	**350–1,000**

Two patterns jump out. Mouse movement dominates whenever the user is active — a third to a half of the heavy column — and it carries the largest payloads. And the spread between columns is an order of magnitude, so any capacity plan built on the "average" user hides a ten-to-one tail. Multiplying by session length, a support-relevant session — the kind a replay is actually opened for — runs three to ten minutes of real interaction and lands in the low thousands of events: roughly **1,000 to 5,000** per session, with pathological long-lived tabs reaching tens of thousands. That range, not the average, is what the platform is built for.

Bytes per event: protobuf on the wire

Event count is half the story; bytes per event is the other. A log point is a protobuf envelope with six fields — timestamp, type, raw `data` bytes, version, order, index — and working one event by hand makes the savings concrete. Take a mouse event whose payload holds two sampled coordinates, say `1024,660,1723687405000|1027,663,1723687405160` (44 bytes):

```
field 1 timestamp (varint)      tag 1 + 6 bytes                  =  7 bytes
field 2 type (enum varint)      tag 1 + 1 byte                   =  2 bytes
field 3 data (length-delimited) tag 1 + 1 length byte + 44 bytes = 46 bytes
field 4 version (fixed32)       omitted when zero                =  0 bytes
field 5 order (varint)          tag 1 + ~2 bytes                 =  3 bytes
field 6 index (varint)          tag 1 + 1 byte                   =  2 bytes
                                          total                  ≈ 60 bytes
```

About 60 bytes, of which the envelope — everything outside the payload — is roughly 16. As JSON, the same six fields must each carry name and punctuation: `"timestamp":1723687405160,"type":"MOUSE_MOVE", "data":"…","order":4123,"index":7.` Keys and braces cost about 90 bytes before a single payload byte, so the JSON event lands near

135 bytes — more than double the protobuf size. In a batch of 200, JSON pays that key tax 200 times; protobuf pays it once, in the schema.

Two payload shapes widen the gap. Payloads that are themselves JSON — scroll state, navigation, form values — must be string-escaped when embedded in a JSON event, doubling every quote and backslash, while protobuf stores them as raw bytes and never escapes. And repetitive structure like a mouse pipe gains nothing from JSON except compression. The honest summary of the worked example: protobuf is roughly two to four times smaller than JSON for typical replay batches, with the biggest wins on the small, repetitive events replay data is full of — which is why the industry line of “three to five times smaller” holds for realistic mixed traffic. It is also cheaper to decode: no string parsing, no escape handling, just field reads. That is why the recorder encodes to protobuf before anything leaves the page, and why the archive and player API stay binary end to end. The format choice is a permanent, schema-wide discount on every byte the platform touches.

From events to sessions

Combine count and size and a session takes shape. A typical session of roughly 2,500 events at an average of about 100 encoded bytes each (mouse above average, terse mutations below):

```
  2,500 events × ~100 bytes  ≈ 250 KB   low-activity session
  8,000 events × ~120 bytes  ≈   1 MB   typical engaged session
 60,000 events × ~140 bytes  ≈   8 MB   heavy, long-lived session
```

Three orders of magnitude between the cheapest and most expensive session is normal. Two things shrink these numbers before storage. Browser uploads carry raw protobuf, but anything on disk is compressed: the receiver zips each event’s payload entering the bus, and the worker re-compresses the reassembled stream into archive chunks. Text-heavy replay data compresses well — coordinates, URLs, attribute names repeat constantly — so the archive is a fraction of the raw stream. And a session is archived once, at the end of its life; the live bytes between browser and archive are a passing wave, not a pile.

Bandwidth: 1,000 sessions through the door

Ingress bandwidth is the first bill a replay platform pays, because every recorded byte is uploaded by a customer’s browser. A useful unit is the thousand sessions, sized from the session model:

1,000 sessions (illustrative)	Bytes per session	Uploaded	Rounded
All light	~300 KB	~300 MB	0.3 GB

1,000 sessions (illustrative)	Bytes per session	Uploaded	Rounded
All typical	~1 MB	~1 GB	1 GB
All heavy	~8 MB	~8 GB	8 GB
Realistic mix	~1.5 MB average	~1.5 GB	1.5 GB

So a thousand real sessions cost the platform roughly one to two gigabytes of ingress, a mid-size customer at ten thousand sessions a month pushes tens of gigabytes through the receiver, and a platform at a million sessions a month receives on the order of 50 GB a day — spread across however many receiver replicas are running. These are model outputs, not code constants, and the correct reaction is to change the assumptions and watch how much the bill moves; that is the value of doing the math. Read traffic is quieter — an archive is pulled a handful of times, usually soon after the session — but processor jobs and every opened replay read whole archives, so chunked, compressed storage matters for egress as much as for cost.

Mousemove, the volumetric enemy

If one event type explains why a replay platform needs throttles, it is mouse movement — the only capture that can fire hundreds of times per minute with no user intent behind it. The raw browser stream fires up to sixty times per second while the pointer moves, which would be thousands of events per minute if each became a log point. The recorder's real constants show the defense: a point is recorded only if it moved at least one pixel, killing micro-jitter; recorded points accumulate for 200 ms and fold into one event with a single pipe-delimited payload; and the batching runs off the main thread. The result: a user who moves the mouse for a full minute produces about 300 mouse events rather than thousands of raw DOM events — an order-of-magnitude reduction before anything leaves the browser, and the most effective filter in the pipeline. Every downstream number in this chapter assumes it. The residual cost is still real: mouse payloads are the largest recurring event type, and a platform that replays the cursor path faithfully chooses to pay for it — the trade is the product value of seeing where the cursor hesitated and rage-clicked. The discipline is to charge mouse data the minimum that preserves that value: coarse sampling, aggressive batching, good compression.

Off the main thread: where batching actually happens

Batching is usually discussed as a bandwidth optimization, but its first beneficiary is the user's own browser. The recorder keeps an event queue on the page and, every 100 ms, hands

it to a **Web Worker** — a second thread with its own JavaScript context — which encodes the protobuf batch, opens the connection, and owns failure handling. The main thread's cost per flush is a few object insertions and one `postMessage`; serialization and network I/O happen elsewhere. That is what makes the recorder safe on a checkout page: even an event storm cannot block the thread that must run the payment handler. The worker also owns retries with real policy: up to **five attempts**, exponential backoff from **one second** doubling to **thirty seconds**, small jitter so thousands of disconnected sessions do not retry in lockstep, and a standing retry timer so a failed batch is not stuck waiting for the next user action. Batching at the edges, in miniature: one flush, one connection, one bounded retry state machine.

Payload budgets: cap what you store

Count and compression are not the only levers; what goes *inside* a payload is the biggest one, because a single ungoverned payload outweighs a thousand mouse events. The recorder's form tracking shows the discipline with real numbers: it refuses `password` fields outright, refuses fields whose names match a denylist (`cvv`, `ssn`, `credit`, `card`), truncates recorded form values to **1,000 characters** (`MAX_VALUE_LENGTH`), and ignores keyboard modifiers. The SDK adds sanitizers that strip sensitive headers and query parameters from network captures before recording. The principle generalizes into a payload budget: decide the largest value worth storing per event type and enforce it at capture, because enforcement is cheapest in the browser. A pasted 50 KB biography costs as much as 500 average events and is never what the replay is for. Truncate long values, redact secrets, drop what you will not replay — every byte stopped at the source skips bandwidth, bus disk, archive storage, and replay egress, and never needs redacting later. Masking is privacy work first (Chapter 7 covers it fully), but it is cost engineering too, and both motives point the same direction.

Storage math at scale

At rest the platform holds two kinds of data with different economics: log points as compressed chunks in object storage, and session documents plus derived analytics in MongoDB. The illustrative model below assumes an average archived session near 1.5 MB on disk, a session document of a few kilobytes, and modest derived data per session (heatmap clicks, backend request records, errors, scroll heat buckets).

Per month (illustrative)	10,000 sessions	100,000 sessions	1,000,000 sessions
New archive data (object storage)	~15 GB	~150 GB	~1.5 TB

Per month (illustrative)	10,000 sessions	100,000 sessions	1,000,000 sessions
Object storage cost at ~$0.01–0.03/GB-month	~$0.2–0.5	~$2–5	~$15–50
`sessions` documents (a few KB each)	~30 MB	~300 MB	~3 GB
Derived collections (errors, requests, clicks, scroll)	~70 MB	~700 MB	~7 GB
Where the money really goes	trivial	noticeable	MongoDB ops
Object storage requests	thousands of PUTs	tens of thousands	hundreds of thousands

Three conclusions. First, at every scale the bulk bytes are cheap: even a terabyte of archive costs tens of dollars a month, because that is what object storage is for. Second, MongoDB never holds log points — only metadata and derived analytics — so its cost tracks document count, indexes, and write hotness, not payload gigabytes; at a million sessions a month the derived collections are gigabytes and operational cost (indexes, backups, sharding conversations) overtakes the raw bill. Third, object-storage *requests* matter as much as bytes: writing one object per event would drown in PUT charges, which is a large part of why the worker coalesces events into chunk archives. Rough by design — redo with your mix and current prices — but the shape is stable.

Serving a session: the chunked archive and the player

Storage layout decides serving cost. The worker does not upload a session as one blob; it re-chunks the reassembled stream into frames and uploads each as a compressed, timestamp-named archive chunk under `sessions/{sessionId}/…`. Chunking buys three things at once: each object stays a sane size, so no single transfer moves tens of megabytes; chunks are self-describing units with timestamps in their names, so a session can be addressed in slices rather than only as a whole; and each chunk compresses independently, so a reader pays decompression only for what it needs. The player API today downloads the session's chunks, reassembles, decrypts, sorts, and streams the log points back in one binary response — simple and fine for typical sizes. The interesting property is what the layout *makes possible*: because archives are chunked and timestamped, a player can fetch only the slice around the scrub position and decode incrementally instead of materializing every event up front. Build the archive chunked from day one even if the first player reads it whole; retrofitting sliceability onto a single-blob format is a migration, while building it in is a layout choice.

Database write amplification

Ask where the event volume touches the database, and the honest answer is: almost nowhere, and that is the point. The receiver never writes log points into MongoDB. Its database work per request is a session lookup plus a throttled metadata update — the receiver skips the update if the last one happened **less than five seconds** ago. A session streaming batches every 100 ms therefore costs the database one small update per five seconds at most, no matter how frantic the user is. Work the counterfactual: a typical session generates about 2,500 events; written individually, that is 2,500 document writes per session, each touching indexes, and at a million sessions a month, 2.5 billion writes — a load that swamps whatever the metadata is for. The design instead confines MongoDB to a few kilobytes of coordination state per session, updated on a human timescale. This is **write amplification** handled by placement: the database sees interaction, not events.

The event volume has to land somewhere, and here it lands on the message bus: the receiver encrypts each log point and publishes it to the session's NATS subject — one message per event, each individually compressed. That per-event cost is real, and the bus is the honest place where it lives in the current pipeline. The worker absorbs it on the other side, pulling the session's messages, decrypting and decompressing, and coalescing the stream into chunk archives, so per-event overhead is paid once on the bus and never on the object store. A future pipeline could batch events into per-session bus messages and move that overhead back to the edges where batching is cheapest; the current one simply sizes the bus for it.

NATS retention versus the S3 archive

Why a message bus *and* an object store for the same events? Because they answer different retention questions. JetStream holds the session's messages from publish until the worker archives the session — minutes to hours normally, longer if the worker pool is backed up. The stream is a **buffer**: durable enough to survive a worker restart, bounded enough not to grow forever, and drained by the worker, which purges a session's subject once the archive upload succeeds (unpurged messages age out by stream retention limits). The object store is the **archive**: it keeps finished, chunked sessions until the retention policy deletes them — 30 days by default in LogNroll's remover job, after which the session is marked `REMOVED` and its objects deleted. The economics follow: the bus is expensive per gigabyte and holds data transiently, so size it for the worst-case backlog; the archive is cheap per gigabyte and holds the long tail, so size it for everything. A platform that kept replayable sessions in its bus would pay bus-grade prices for archive-grade retention; one that archived without a buffer would lose sessions whenever the worker hiccuped. Retention is two decisions: how long a failed archive may sit in the buffer, and how long a finished session deserves to live in the archive.

Where a tenfold traffic spike lands

Take an illustrative platform running at 100 sessions started per second and multiply traffic tenfold for an hour — a promo, a launch, the shape of Candlewood's Shelf Saturday in Chapter 9. Each tier absorbs a different fraction of the shock, and knowing which is which separates a busy hour from an incident. The **receiver** sees the full blast: ten times the requests, each needing parsing, session lookup, per-event encryption, and bus publish. Receivers are stateless — their only shared state is MongoDB and NATS — so the fix is horizontal: more replicas, load-balanced traffic, and a timeout path that answers an honest 503 before the edge proxy gives up. Receiver CPU is the spike's first victim and easiest cure. The **message bus** is the shock absorber: JetStream takes the burst onto disk and lets the worker pool consume at its own pace, so a sustained spike shows up first as a growing backlog on bus disk. A stream sized for minutes of worst-case backlog turns a traffic spike into a storage spike the workers chew through afterwards; one sized for average load turns it into dropped sessions — the most important capacity decision in the architecture, invisible at average load. **MongoDB** barely notices the event-rate spike: thanks to the five-second throttle its write load tracks active sessions, not events, and even a session-count spike is a linear, predictable rise in small document updates; derived analytics are written later, at processor pace. **Object storage** is most insulated: writes happen only when the worker archives a finished session, so a spike that inflates events but not sessions barely moves S3 traffic, and even a session spike triples a cost that is already cheap per unit. The ordering is the design: put bursty, unpredictable load against tiers that scale horizontally or buffer on disk, and steady load against tiers that are expensive to scale. Getting it backwards means fighting the database during the spike instead of letting the bus eat it.

Three design principles

The worked examples reduce to three principles, and nearly every decision in the pipeline is one of them applied.

Move data as few times as possible. Every hop — browser to receiver, receiver to bus, worker to archive, archive to player — is a copy with CPU and latency attached. The pipeline minimizes both the hops and the size of what travels: events go browser to bus to archive to player and nowhere else, log points never detour through the database, and each hop forwards data in a shape it need not transform.

Compress early. Bytes kept small stay small through every later hop, so compress as close to the source as the format allows. Protobuf is the first compression, a shared schema that removes the per-event key tax forever; real zipping happens at the edges, per event entering the bus and per chunk entering the archive, and replay text compresses well because it repeats. The caveat visible in the real pipeline is granularity: compressing each tiny event

separately adds per-message overhead a later stage must undo, while chunk-level compression is where the archive actually gets small.

Batch at the edges. The fewest, largest round trips win: the recorder batches 200 events per 100 ms flush, the database is updated at most every five seconds per session, and the worker coalesces whole sessions into chunk archives. Batching is cheapest where data is created and most valuable where each individual write is expensive — which is why the browser and the database are batched hard, and why the one place the current pipeline accepts per-event cost, the message bus, is designed to absorb it.

None of these principles are exotic. They are this chapter's arithmetic generalized into habit, and they are why a platform recording thousands of events per session can store a month of replays for pocket change per customer while answering any of them in a second.

Chapter takeaways

- One active user minute yields tens to hundreds of events and a typical session one to five thousand; mouse movement is the largest and most controllable slice of that volume, tamed by the real 200 ms batching window and one-pixel filter.
- Protobuf's worked-example advantage over JSON is a two-to-four-times size reduction on typical batches, largest on the small, repetitive events replay data is full of, plus cheaper decode and no per-event key tax.
- Bandwidth and storage bills are model outputs, not surprises: a thousand sessions is roughly a gigabyte of ingress, and a million archived sessions a month costs on the order of tens of dollars in object storage while MongoDB cost tracks document count and hotness, not payload bytes.
- Real throttle constants — 100 ms flushes, 200-event batches, the 1,000-character form cap, the five-second session-document update cadence — are the platform's cost controls, each removing an order of magnitude somewhere downstream.
- Write amplification is managed by placement: events never touch the database, per-event cost is accepted only on the message bus, and the worker coalesces sessions into chunked archives so object storage sees whole sessions, not individual events.
- A tenfold spike lands hardest on the receiver, which scales horizontally, and the bus, which buffers on disk, while MongoDB and object storage barely notice — because bursty load is routed to tiers built to absorb it.

Chapter 20: Where This Is Going: Future Improvements

A book about how a working system is built should end by admitting that the system is still being built. The architecture you have read about is not a finished monument; it is a live platform, and the team behind it is mid-stride on several of the changes in this chapter right now. This final technical chapter is a roadmap, and roadmaps need an honesty disclaimer: they are directional, not contractual. Some items are real work already in progress. Others are natural next steps any engineer could propose tomorrow. A few are speculative, where the industry itself has not yet converged. The chapter marks each item with its level of reality, because telling “we are doing this” from “this seems worth trying” is exactly the discipline this book has been teaching.

The roadmap has a logic worth naming. Session replay platforms sit at the intersection of three pressures: more data (longer, richer sessions), more privacy obligation (the data is someone’s actual behavior), and more intelligence (the data is useless unless something can make sense of it). Every item below is one of those pressures arriving at a specific component; read it that way and the proposals cohere into one direction of travel.

Finish the Java-to-Go migration

The most concrete item on this roadmap is also the furthest along. LogNroll began as a Java platform: Spring Boot services for the receiver, the worker, the player API, and a housekeeping archiver, sharing MongoDB, NATS, and S3. A migration to Go is in progress, service by service. The receiver, the worker, and the player API now have Go rewrites; the development ingress already routes traffic to the Go services while the production Helm chart still targets the Java ones; and the old Java archiver has already been split into two Go cron jobs — a session-status job that runs every minute and a session-remover job that runs hourly.

Why it matters: the worst state of any migration is the middle. Two receivers, two workers, and two player APIs mean two of everything operational — two memory models, two deploy stories — and development and production behave differently until cutover finishes. Finishing means one runtime per service, one set of images and probes, and the freedom to delete the legacy Java codebases, which remain a second source of truth for the same contract. The rough cost and benefit: the expensive part, rewriting the busiest services, is already spent, so what remains is mostly production cutover and deletion — low cost, with benefits that compound as later roadmap items land on one runtime instead of two.

One contract, shared: consolidating duplicated packages

If you read closely across these chapters, you may have noticed a recurring phrase: “identical copy.” The `LogPoint.proto` contract exists as identical copies in the logger, the receiver, the worker, the session processor, and the player API — and, as generated code, in the Angular front ends too. The shared AES encryption-key handling, and even the logic selecting which of the two S3 tenants a session belongs to, is copy-pasted across the services that need it. This duplication is deliberate — LogNroll’s repos are independent by design — and it has served the platform well. But it is a growing maintenance risk: a contract change means changing several repos in lockstep, and a field added in one copy but not another fails only at runtime, between services.

Why it matters: the data contract is the platform’s most important interface, and it currently has no single owner. Consolidation means publishing the contract once — a canonical `.proto` feeding a code-generation pipeline that produces the Go, Java, and TypeScript bindings every service and front end imports — and extracting the shared encryption and S3-selection logic into a small shared package per language. The rough cost is real but bounded: a shared module, a build step, and a coordinated cutover of the consuming repos. The benefit is structural: contract changes become one edit plus regenerated code, cross-repo drift becomes a compile-time error instead of a production surprise, and the duplication story in this book becomes a historical footnote rather than a permanent design tax.

Versioning discipline: making PROCESS_VERSION a habit

The platform already contains a versioning mechanism, and the scar tissue that proves it necessary. Log points carry a `version` field on the envelope, and the session processor tracks a `PROCESS_VERSION` per analysis: when a processor’s output semantics change, the version must be bumped, because the pipeline skips reprocessing any session whose recorded `applied` version is already at or above the current one. Miss the bump and sessions silently keep the old shape of derived data; readers that expect the new shape gate on version — the scroll heat repository filters for version two or higher — so an un-bumped processor quietly serves stale analytics. That asymmetry is the gotcha: versioning protects you only if bumping is part of the definition of “changing something.”

Why it matters: replay platforms accumulate derived data — errors, backend requests, heatmap clicks, scroll attention — whose shape changes as analysis improves and which outlives the code that wrote it. A deliberate discipline turns the ad hoc field into a contract: an explicit registry of versions per output, a policy that every semantic change ships with a bump, and ideally a CI check that fails when a proto or processor change arrives without one. The cost is process, not machinery — the machinery already exists. The benefit is trust:

derived data is labeled with the shape it holds, and reprocessing is triggered deliberately instead of discovered by accident.

Encryption: modern AEAD, per-tenant keys, rotation

Encryption today is symmetric AES with a single shared key: the receiver encrypts event payloads at ingest, and the worker, processor, and player API all decrypt with the same `encryption.key`, distributed as a shared secret. It is simple and right for encrypting at the edge without adding hot-path latency. The natural next step is not a criticism of that baseline but an evolution of it. The industry has moved toward authenticated encryption — modern AEAD modes, which detect tampering as well as conceal content — and toward envelope schemes in which each tenant is encrypted with its own data key, and data keys are wrapped by a small set of master keys. That structure makes rotation an operational routine: keys are versioned, payloads record which version encrypted them, and rotating means minting a new data key for new writes while old data remains readable until it ages out of retention.

Why it matters: the blast radius of a single shared key is the whole platform. If it leaks, every tenant's archive is exposed at once, and rotating it means re-encrypting everything or carrying the compromise forward. Per-tenant keys shrink the blast radius to one customer and make "we rotated your tenant's key" an answer to an auditor's question. The costs are a key-management layer, key-version tracking, and a migration path for stored payloads; the benefit is a posture that matches the data's sensitivity. Nothing about capture changes — encryption stays at the edge — so the client story remains as written.

AI-assisted session analysis is already arriving

The most surprising item on this roadmap is the one already shipping, quietly, as a developer tool. LogNroll has built an MCP server — a Model Context Protocol server, in Go, that gives LLM-powered chats and agents direct access to session data. Connected to a capable assistant, it can list a company's sessions with user, device, page, and status filters; pull the raw recorded events of a session — the same decrypted log points the player replays; inspect detected errors, network requests, heatmap clicks, and scroll attention. It reads the same MongoDB collections and S3 archives as the player API, and it applies the platform's own authorization model, scoping every tool to the requesting user's companies. Payloads are sized for an AI context window — events over 8,192 bytes are truncated with a marker rather than silently dropped.

Why it matters: session data has always been locked inside a proprietary player; the only way to ask questions of it was to watch it. An MCP server turns the archive into something a general assistant can interrogate: "Summarize what this user did in the two minutes before

the error," "Did the failed request retry, and what did the second attempt return?" The cost is small — model calls and latency are cheaper than a human watching minutes of replay — and the benefit is the direction made concrete: the same data, finally readable by software that can summarize it. The privacy-preserving variant is visible in the same design: run the tool against a local model, and session bytes never leave the machine. Powerful analysis with no data exfiltration is likely the shape of debugging for the next decade, and replay platforms are positioned for it because they already hold the data in a structured, queryable form.

Live co-browsing and streaming sessions

Everything so far has been about the past: record, store, replay. The most requested future feature is the present tense — watching a session while it happens, as when a support agent sees the customer's cursor move right now and guides them through the checkout. The difference from replay is architectural: replay reads finished archives, while co-browsing needs a live path from the browser to a viewer with end-to-end latency in the low seconds. The platform already has most of the plumbing — the event stream, the bus, the player — but the live path is a new tier: a streaming channel from the bus to the viewer's player, ordering rules that tolerate a moving front edge, and honest degradation when the connection stalls.

Why it matters: the highest-value support moments are the live ones, and "can you see what I see" has been support's fantasy since the telephone. The cost is a real-time layer with its own scaling and reconnect story, plus privacy work — live viewing is live exposure, so masking must be as disciplined as in replay, ideally with an explicit "you are being watched" indicator. The benefit is support that closes tickets in minutes instead of email round trips — the outcome the Candlewood story keeps circling.

A faster player: WASM decode and instant seek

Replay quality is a front-end performance problem, and its cost concentrates in long, event-dense sessions: decoding tens of thousands of log points, sorting them, and rebuilding the page — work that today happens in JavaScript in the browser. The roadmap directions move that work out of the way of the human scrubbing the timeline: decode protobuf in a Web Worker or in WebAssembly, as the recorder's own worker architecture already demonstrates; build timestamp indexes over the chunked archive so seeking to a moment is a lookup instead of a linear scan; and virtualize the timeline so the player renders only the window the viewer is in rather than materializing the whole session. None of this changes what a replay is — only how fast it answers.

Why it matters: every millisecond of scrub latency is friction between a support agent and the answer they are hunting. Short sessions are fine today; the long, heavy sessions replay exists

to debug are where decode and seek speed become the product. The cost is front-end engineering and the complexity of a second decode path; the benefit is a player that feels instant even on sessions measured in tens of thousands of events — the difference between a tool people use and a tool they open reluctantly.

Privacy automation

The privacy chapter described the current state: capture only what you need, mask sensitive fields by type and name, sanitize network payloads, and let retention do the forgetting. The automation frontier is making those rules smarter and cheaper to maintain. Today the recorder's denylist approach is honest and brittle — it knows `password`, `cvv`, `card`, and friends, but not the field your customer calls `account-number-2`. Model-assisted redaction would recognize sensitive content by shape and context: a sixteen-digit number in an otherwise innocuous field is probably a card number; the same digits inside a product code are not. That capability can apply at capture, where a classifier decides what not to record; at ingest, where a redaction pass cleans anything that slipped through; or in the player, where sensitive regions are blurred for viewers without permission to see them.

Why it matters: privacy rules are only as good as their coverage, and hand-written denylists guarantee blind spots. Context-aware masking is the difference between "we mask the fields we thought of" and "we mask the things that are actually sensitive," and it serves the platform's license to operate: less sensitive data stored means less to leak or explain. The cost is real — models at capture latency, false positives that hide legitimate fields, human review — but it is the same cost every privacy-conscious vendor is paying, and replay platforms pay it earlier because they capture the most. Automated redaction also composes with the rest of the roadmap: redact at the edge, and the AI analysis tools earlier in this chapter inherit clean data for free.

Finer cost controls and retention

Storage is the platform's own cost of goods, and retention is the dial that controls it. Today the platform keeps finished sessions for 30 days by default — a sane default for incident hunting — and retention is a product decision per plan. The roadmap makes the dial finer: retention tiers chosen per plan or per company; storage accounting that shows a customer how many gigabytes their sessions occupy and what they cost; quotas and alerts before a runaway library surprises anyone; and a middle path between keep-everything and delete — coarsening old sessions so metadata, errors, and analytics survive while pixel-level event data ages out, leaving a degraded but searchable record. Replay data is bulkier and more personal than logs, so the economics of memory are a product surface, not just an internal metric.

Why it matters: retention is where the customer's budget meets the platform's cost, and a one-size default forces a choice between too much data for a small shop and too little for an enterprise under audit. Finer controls let each customer pay for the memory they actually use — fairer, and better business. The cost is billing and lifecycle complexity, since every new retention rule is a new state in the remover job; the benefit is a cost story that scales from a bookstore to a bank without either subsidizing the other.

Enterprise needs: SSO/SAML, audit logs

Session replay data is some of the most sensitive operational data a company owns — actual customer behavior, often including payment-flow interactions — so the enterprise sales conversation turns quickly to access control. Today the platform authenticates with magic-link email codes and issues JWT tokens. The enterprise roadmap is the standard suite on top of that base: SAML or OIDC single sign-on so access follows the corporate identity provider, SCIM for automated user provisioning, role-based access refined beyond the current team model, and audit logs recording who opened which session and when. That last item deserves emphasis: viewing a session is a privacy-relevant act, and "who looked at this recording" is a question every compliance officer will ask — the answer should not require digging through application logs.

Why it matters: none of this changes the core product, but all of it changes whether the product can be bought. Procurement checklists are gatekeepers, and SSO and audit trails sit near the top of every one. The cost is integration work — identity providers, audit storage, legal review of retention and residency commitments — and the benefit is access to deals where replay data is sensitive enough that the buyer needs accountability before analytics. The authorization pattern is already proven inside the platform: the MCP server scopes every tool call to the requesting user's companies, and generalizing that discipline into first-class audit is a smaller step than it looks.

Edge and regional ingestion

Latency and residency both point the same direction: closer to the user. Today the recorder posts to a single public receiver endpoint, and the platform's regional story is limited to storage — the two S3 tenants, one in Frankfurt and one in Amsterdam, show that the code already selects a region per session. The roadmap extends that selection backward toward the browser: regional receiver endpoints so a European session lands in Europe and an American session in America; the recorder bundle and receiver addressable from the edge closest to the visitor; and data residency as an explicit promise — sessions ingested in a region are stored and processed there, so a customer with residency requirements can point

to the routing table, not a policy document. The hot path is stateless by design, which makes regional fan-out a routing problem rather than an architecture rewrite.

Why it matters: two different customers are asking for this. One wants speed — a distant receiver adds latency to every batch flush and every replay open. The other wants geography — EU data that must stay in the EU, stated in a contract, not implied. The cost is per-region infrastructure and running the pipeline in more than one place; the benefit is a platform that can say where its data lives, which is becoming a purchase criterion on its own.

An SDK ecosystem

Recording today lives in one place: a JavaScript recorder embedded in a web page, wrapped by a small framework-agnostic SDK, with a Chrome extension for manual injection. The ecosystem roadmap spreads that capability to where users actually are. Framework plugins — React, Vue, Svelte — would ship batteries-included defaults: network sanitizers preconfigured, masking presets for common form patterns, and error-boundary hooks so a component crash is captured with its replay. Mobile webviews are the bigger gap: much of the modern web runs inside native apps' embedded browsers, invisible to a pure-web recorder, so bridging the recorder into iOS and Android webviews — and eventually adding a native mobile capture layer for touches and gestures — would record sessions that today are simply dark. A native mobile SDK does not exist yet.

Why it matters: every framework and platform where the recorder does not run is a blind spot for the customer's support team, and the web is fragmenting into frameworks while migrating into app webviews. The cost is a maintenance surface that multiplies with every supported target, since each plugin is a small, permanently supported product; the benefit is coverage — more sessions recorded, more of the user's journey visible, and fewer sessions ending at the edge of a webview.

Open formats and industry convergence

The last direction is the industry's, not just one platform's. Session replay today has no universally shared format the way logs have JSON and traces have W3C conventions. Projects like rrweb have demonstrated open, community formats for DOM-based recording, and vendors are slowly converging on the idea that a session should move between tools the way a log line can. For a platform with its own battle-tested protobuf contract, the interesting question is where openness pays: at the edges, where capture formats the community understands lower the barrier for developers, and in the schema itself, which LogNroll already publishes in this book's appendix and shares verbatim across every language in its pipeline — a de facto openness that a formal open format would extend rather than replace.

The tension: proprietary formats are how replay vendors differentiate, and standards move slowly.

Why it matters: formats are lock-in, and lock-in is the one objection every evaluation of a replay platform eventually raises. A path toward open, portable sessions — export a session, hand it to another tool, keep the recording when you leave — removes that objection and grows the whole category, the same way open log and trace formats grew observability. The cost is mapping an optimized internal format onto a general one, and the risk of betting on a standard that shifts; the benefit is a market where session data, like log data before it, belongs to the customer who recorded it. Whatever the timeline, replay formats are converging on openness that outlives any single vendor.

The shape of the roadmap

Read together, these items describe a platform stretching in three directions at once: finishing what it started (the migration, the consolidation), hardening what it holds (encryption, privacy, access, residency), and making the data more useful faster (AI analysis, live sessions, a faster player, open formats). The first group is cost, the second trust, the third value, and a healthy roadmap needs all three — cutting costs alone stalls, building trust alone starves, and adding value without hardening leaks. The reference implementation in this book is a real system with real tradeoffs, and its future is the same negotiation every reader's systems will face: how much to spend on the foundation, how much on the lock, and how much on the view.

> **Story checkpoint — Candlewood Books:** Two years later, in 2028, the ritual inside Candlewood's support queue has changed. Before anyone opens a replay, they read the AI summary that LogNroll writes for every flagged session — a few lines: the user's path, the error, the moment of rage-click — and only then, when the summary says something strange, does a human press play. Marta's team resolves the routine cases from the summary alone, and the replay has become what the summary says it is: the place you go when you need to see the truth with your own eyes.

Chapter takeaways

- The Java-to-Go migration is real and mid-flight: the receiver, worker, and player API have Go rewrites, the old Java archiver is already two Go cron jobs, and what remains is production cutover and deleting the legacy services.
- The duplicated contract, encryption-key handling, and S3-selection logic are deliberate but costly; the roadmap is one canonical contract with generated bindings and shared packages per language.

- The `PROCESS_VERSION` mechanism enforces reprocessing discipline only if bumping the version is part of every semantic change; version gating in readers keeps derived data honest.
- Encryption's next step is authenticated AEAD modes with per-tenant data keys and routine rotation, shrinking the blast radius of a single shared key without touching the capture path.
- AI-assisted session analysis is not speculative: an MCP server already exposes sessions, errors, heatmaps, and network data to LLM agents with per-user company scoping, and local-model analysis offers a privacy-preserving way to debug sensitive sessions.
- The rest of the roadmap — live co-browsing, a faster player, privacy automation, finer retention, SSO and audit, regional ingestion, an SDK ecosystem, open formats — is directional, each item one of three pressures: cutting cost, building trust, or making the data more useful.

Chapter 21: Epilogue — and Should You Build or Buy?

Two years later

Rain ran down the windows of the office above the shop again, and the espresso machine coughed once, which everyone agreed was its way of noting that it was Monday. It was December 2028, the third holiday season since the Summer of Blame, and Marta Reyes opened the support queue and found nine tickets: nine, for a whole holiday weekend, the busiest season a small bookstore could have. In the bad old days, a weekend like this one would have drowned her, and every ticket would have been a confession of blindness: *It charged me twice. Nothing happens when I press Pay. The confirmation email never comes.*

Those complaints had not vanished so much as been retired. They surfaced now as variants — specific, fixable — and each arrived with the answer half-attached. “Checkout stuck on the gift-card step,” read a ticket from that morning, and the customer had done something nobody did back then: she had included a link. *I watched my own session and it stops right here, at 0:42.* Marta clicked, watched 42 seconds of a stranger’s careful December shopping, saw the gift-card field refuse the code, and replied within six minutes. The old average had been 40, most of them spent asking questions the customer could not answer.

Beside her, a seasonal hire named Elise, three weeks in, was watching a “December classics” playlist: recorded sessions from past holidays, anonymized and masked, each opening with a summary line the platform’s newer AI-assisted tools had written, then the session itself, the cursor moving through the old checkout like a ghost. Marta walked her through the vocabulary as she had once walked new hires through the returns spreadsheet: dead click, rage click, device chain, masked field, network timeline, scroll depth. “What you’re looking for,” she said, “is never what the customer says it is. It’s what you can see them do.”

That was the whole of the change. Candlewood had not become a company of surveillance experts; it had become a company that spoke the language of seeing and taught it to everyone who answered the phone, packed the boxes, or wrote the code. When a new hire asked the old question — *which browser was it, whose fault is that?* — the answer had become a ritual: pull up the session. Look. The phrase that began as Maya’s joke in 2026, “we never argue about the browser anymore,” had become the shop’s actual policy, written on the whiteboard by the espresso machine. They did not argue about the browser because they did not need to. They looked at it.

The library that stayed

The handwritten note arrived in late October, addressed to Maya in a careful, upright hand she recognized before she opened it: a single sheet of Northside Public Library letterhead and two paragraphs in blue ink. Dana Whitfield was retiring at the end of the year, and she wrote to say so, and to say something else.

In 2026, when the portal kept eating our carts, I told you we would move the account if you couldn't fix it. I remember writing that email and meaning it. I'm glad we never had to send the second one. The summer order has gone in on time for three years running, my staff stopped calling me about the portal sometime in 2027, and I've watched a small shop learn to see its own customers. Whatever you're doing, keep doing it. — Dana

Maya read it twice, then read it aloud at the Tuesday meeting, and nobody spoke for a moment, because the letter carried a whole history in a few sentences. The account that had nearly walked out in May 2026 — whose staff could not add a book to a cart on a narrow Safari window with a cookie banner in the way — had not only stayed; it had grown into a standing agreement with three branch systems, and the summer order now ran to more than a thousand titles. Marta, who had solved the original cart mystery in minutes with Dana's consent while watching the portal session replay, kept the note in the top drawer of her desk. The reason was not a technology. It was that Candlewood had learned to see what Northside's staff saw, and to fix it before the library had to describe it.

The redesign that paid for itself

The homepage redesign Candlewood shipped in the summer of 2026, ahead of the holiday rush, was the first real test of whether replay data could do more than debug. Priya argued for it with evidence that would have been unthinkable a year earlier: scroll-heat maps showing where mobile shoppers stopped reading and, more importantly, where they never went; click heatmaps with their patient clusters of near-misses on the search box; session after session of people scrolling past the Reading Room offer entirely. The new homepage was designed by people who had watched a thousand real visits: the subscription offer moved up where mobile visitors could reach it, the gift section stopped hiding below the fold, search got the suggestions it had always needed, and the rage-click cluster on the search results page quieted to nothing.

The payoff showed up that first December and held every December since: checkout completion up 18 percent over the year before the relaunch, December ticket volume down roughly 70 percent from the nightmare spring, support time per ticket down from about 40 minutes to about six. Maya was careful, in the talks she sometimes gave to other small retailers, to describe those numbers for what they were: what happened to one shop, not a

promise. Privately she kept the numbers that made it real — the confirmation email arriving on time because failed emails had been found in replays instead of in complaints; Mrs. Alvarez, who still ordered her Reading Room box every month and had not called about a payment since 2026.

The redesign itself was never finished, and that was the point. The heatmaps and scroll-reach data came back before every season, and the team made small changes the way a gardener tends a bed: a button enlarged because a regular customer's session showed her squinting at it, a form shortened because mobile visitors abandoned it at the third field. Replay had become a way of seeing — consulted routinely, respectfully, and before anything broke.

Maya's letter

The letter Maya wrote that winter was to the LogNroll team, who had asked whether they could tell Candlewood's story in a book about how session replay worked. Maya said yes on one condition: that the story be told whole, with the parts where they had blamed the browser, the bank, and the customers left in. A story that started at the happy ending would teach nobody anything.

Then she sat down to explain what their machine had actually done. It took three drafts. The first was full of numbers; the second, full of thanks. The third, the one she finally sent, ended with a thought she had carried since that rainy Friday morning when she watched a stranger press a button that would not open:

We thought we were buying a way to fix our website. What we actually bought was a way to see our customers — really see them, not count them. And I've learned that seeing is where caring begins. You cannot fix what you will not look at, and once you have looked at a person's trouble — watched them try, watched them wait, watched them give up and come back and try again — you will not rest until you have fixed it for them. That is the whole secret, if there is one. Thank you for the seeing.

Much later, when the LogNroll team wrote the acknowledgment that opens this book — to the fictional Maya Okafor, whom they made up but have met a hundred times, and to every support agent who ever asked to see a customer's screen — they found they were answering her letter. Seeing is the beginning of caring about the details; for the reader who has come this far, that is the thesis of everything between these covers. For the engineer deciding what to do next, it is also the question — because if seeing is that powerful, the temptation to build the seeing yourself is almost irresistible.

Should you build or buy?

Here is the question most engineers reach by the end of a book like this one: *I could build this*. And it is true — you could; that is the point of the previous twenty chapters. The question is whether you should. The honest answer: sometimes yes, often no, and usually something in between — if you keep asking.

Both decisions have killed companies. Teams have bought a replay service and starved it of configuration and training, ending with a tool nobody trusted and a subscription nobody remembered to cancel. Teams have built their own pipeline and discovered two years in that they were maintaining a product — browsers change, privacy law moves, player parity is a treadmill — instead of shipping what they sold. Neglect or drift: pick your failure mode.

When building makes sense

Building makes sense in a specific set of situations. The first is privacy and data control so strict that no third party may come near the sessions: on-premises or offline-first deployments, air-gapped environments, regulated industries, or a data-residency requirement no vendor meets. If your sessions must never leave your perimeter, the question answers itself — you run the recorder from your own domain, receive events on your own receivers, archive them in your own object storage, replay them from your own player, and operate every retention and consent obligation yourself.

The second is deep customization: exotic surfaces such as complex canvas interactions, embedded applications, or webviews that behave like native apps, where a general-purpose service will always compromise. If replay data feeds your own analytics in unusual shapes, or the recorder becomes part of your product rather than a lens on it, the build starts to look less like infrastructure and more like differentiation. The third is that you may already be most of the way there — a team with protobuf, a message bus, object storage, and a worker fleet has built the boring half of the pipeline already. The fourth reason rarely appears in cost models: learning; many products now sold as replay services began as an internal tool that escaped.

What building actually costs

Now the honest part, where most build decisions die. Building means building *everything this book covered* — not the parts you find interesting, all of it. You will write a recorder that captures a faithful, privacy-safe event stream and never breaks the page it rides in on. A receiver that absorbs your traffic at its worst moment without dropping a batch. A durable bus between ingestion and archival. Workers that claim, compress, and file every finished session into cheap storage, and the lifecycle machinery that ages sessions from active to

finished to removed, because retention is a legal promise, not a preference. A processor that turns raw events into the errors, slow requests, and heatmaps your team will use, plus background jobs and monitoring. A player API and a player — and the player, the DOM reconstruction from mutation events, is the hardest software in the pipeline. Then the product shell and operations: session lists, digests, access control, scaling, quotas, on-call discipline.

Read that list again: it is the twenty-one chapters you just read. Building means becoming the maintainer of every one of them, forever. Browsers ship quarterly, and your recorder must keep pace or your data silently degrades. Privacy regulation moves, and your masking, consent, and retention machinery must move with it. Add the accounting nobody puts on the whiteboard — on-call for a system that ingests every user interaction, security review of a component that handles keystrokes, data-controller obligations for sessions that are records of real people — and the honest total is a product line, whose cost never reaches zero.

Open-source foundations change the arithmetic but not the conclusion. Libraries such as **rrweb** give you a mature DOM recorder and reconstruction player, and you should use them rather than write your own from scratch. But a recorder and a player are two components of the twenty-one chapters. The ingestion gateway, the bus, the archival workers, the lifecycle machinery, the processor, the access control, the privacy pipeline, the operational discipline — none of that comes in the library, and that is where the years go. If you build, build on rrweb. And still count the full cost.

When buying makes sense

Buying — or mostly buying, or renting while you learn — makes sense for everyone else, which is most teams. The first reason is time to value: a session replay service is a snippet of JavaScript this afternoon, while a self-built one is a roadmap with quarters attached. If your checkout is leaking customers right now, the time to build a recorder is time your customers do not have. The second is focus: replay is a lens on your product, not your product — unless your product is replay. The third is that the hard problems are somebody's full-time job: masking, consent, retention, scaling, and player parity, maintained by people whose working life is browser quirks and privacy law. Buying is how you hire that expertise for a subscription instead of a payroll line.

Buying deserves the same honesty as building; it is not zero work. Choose a vendor on privacy posture, data handling, retention, and residency as carefully as you would choose a build. Configure masking and consent before you record a single real user — the work of Chapter 7. Train your support team to read sessions — the work of Chapter 18. The failure mode of buying is not the subscription; it is buying and then not doing the work of seeing —

turning on the recorder and never watching a session. Candlewood's story is what buying looks like when the buyer does the work.

The middle path

There is a third answer, and for a surprising number of teams it is the right one: start with a service, grow into a hybrid, and revisit the decision every year. The hybrid shapes are many: keeping your own archive of the session data you care about while a service handles general replay and analytics; recording with an open format for the funnels that matter most while the service covers the long tail; self-hosting the recorder bundle while a vendor runs the backend. The common thread is the discipline: an annual review, written into the calendar like a security audit, built on four questions. Has our privacy or residency requirement changed? Has our need for customization crossed the line where a vendor's compromise costs more than a build? Has our traffic grown enough that the economics flipped? And — the question nobody asks — is our team still excited about the thing we actually sell?

Candlewood ran that review every June, and each year the answer was the same: its replay service was a way of seeing it could not have maintained itself. But Maya kept the question on the agenda — and that, not the answer, was the discipline that mattered.

The architecture, in ten lines

Before the final word, set down the whole machine in prose, because the recap is what you should carry out of here. A recorder inside your page turns a visit into a stream of small events — clicks, scrolls, keystrokes, DOM changes, console messages, network calls — batched, compressed, and sent in a compact binary encoding. An ingestion gateway receives them at the edge, tracks the session in a coordination store, encrypts the payloads, and publishes them to a durable message bus. Workers claim finished sessions off the bus, decrypt and re-chunk the streams, and file them as compressed archives in cheap object storage, where a lifecycle job retires them after a promised retention window. An offline processor reads the archives and distills the insight — errors, slow requests, click heatmaps, scroll reach. When someone opens a session, a player API fetches and decrypts the archive and streams the events to a browser player, which rebuilds the page and replays the visit as if it were a film. And wrapping the journey are the commitments that make it ethical: sensitive fields masked before they leave the browser, payloads encrypted in transit and at rest, access scoped to the session's own company.

Where to go next

If you are leaving this book toward a decision, the appendices are your toolkit: the protobuf contract in Appendix A, the storage and collection reference in Appendix B, the configuration reference in Appendix C, and the glossary in Appendix D when the vocabulary blurs. Appendix E points onward — to the LogNroll documentation and engineering blog, to the NATS and Protocol Buffers references, to rrweb and the open replay ecosystem. Deciding whether to build? Reread Chapter 9 and Chapter 16: the receiver is where hot-path discipline lives, and the player is where the years go. Buying? Reread Chapter 7 and Chapter 18: masking and consent first, training always. Whichever way you lean, the cheapest experiment is the one Candlewood ran: turn on a service and watch ten real sessions of your own product.

The final word

Every chapter of this book has described a machine for seeing: a recorder that watches, a pipeline that carries, an archive that remembers, a processor that understands, a player that shows. It would be easy to come away impressed by the engineering and miss the point of it. The point is the person at the other end — the customer pressing a button that will not open, the librarian assembling a summer order, the seasonal hire learning what a rage click looks like before it costs a sale. Replay makes you see those users. And seeing, as a small bookstore in the rain discovered, is the beginning of caring about the details. Not the end — caring is work, and the work is fixing what you see, respecting the people you watch, and keeping the recordings only as long as you promised. But it begins with seeing, and now you know how the seeing is built. Go watch a session. Then go fix the thing you see.

Chapter takeaways

- Two years on, Candlewood's retained library client, its heatmap-driven redesign, and its replay-fluent support team turned "we never argue about the browser anymore" from a joke into policy.
- Replay stopped being a debugging tool and became a habit of seeing — consulted before every season, used to coach seasonal staff, and applied to redesigns before they shipped.
- Maya Okafor's letter is the seed of the book's dedication: replay makes you see users, and seeing is where the caring begins.
- Building your own replay means building all twenty-one chapters of this book — recorder, receiver, bus, workers, storage and lifecycle, processor, player, privacy, and operations — and maintaining them forever.

- Building makes sense for on-premises or offline privacy needs, deep customization, teams that already run most of the stack, and genuine platform ambitions.
- Buying makes sense when time to value and focus dominate; its failure mode is neglect, not subscription cost.
- The middle path — start with a service, grow into a hybrid, review the decision every year — fits most teams, provided the review actually happens.
- Libraries like rrweb remove the hardest two components from the build; the rest of the pipeline is where the years go.

How Session Replay Service Works — first edition, September 2026. Written by the LogNroll team and released free under CC BY 4.0; share it with attribution. Download the PDF and EPUB, and read more, at lognroll.com/book.