ANNEXOPS · AI COMPLIANCE INFRASTRUCTURE

EU AI ACT · ARTICLE 12 · SUPPORTING EVIDENCE

AnnexOps Runtime Logger — Integration Guide

Engineering setup guidance. Not legal advice, and not a certification of conformity.

AnnexOps Runtime Logger — Integration Guide

> This guide is engineering-facing setup guidance, not legal advice. AnnexOps does not certify conformity — keeping a tamper-evident runtime log **supports** (but does not by itself discharge) your EU AI Act Article 12 record-keeping obligations. What Article 12 requires of your system, and whether this record satisfies it, is a determination for you and your counsel.

Keep a tamper-evident, per-system log of every inference your service makes — evidence that an exact input/output occurred, without ever handing us the content itself.

You send **hashes only**. The raw prompt and response never leave your infrastructure.

How it works

```
your service ──(SHA-256 at source)──▶ POST /v1/runtime-logs ──▶ 202 Accepted
                                                                    │
                                                  queue ──▶ worker ──▶ tamper-evident
                                                                       hash chain per
                                                                       (org, system_key)
                                                                         │
                                                  Vault ▸ Runtime Logs ──▶ verify chain
                                                                           Export Art 12
record
```

1. For each inference, your code computes the SHA-256 of the **input** and the **output** at source.

2. It POSTs a batch of these hashes (plus lightweight scalars — model version, confidence, latency) to `/v1/runtime-logs`, authenticated with an org **API key**.
3. The endpoint returns **202 Accepted** immediately and enqueues the batch.
4. A worker appends each event to a **per-system hash chain** — every row links to the previous one, so any later tampering is detectable.
5. In the app, you map each `system_key` to a registered AI system, then verify the chain and export the Article 12 record.

Privacy guarantee: only `sha256(input)` and `sha256(output)` — 64-character hex digests — plus scalar metadata are transmitted and stored. AnnexOps never receives, stores, or logs your prompts or responses.

Before you start: mint an API key

1. Go to **Settings → API keys → Generate key**.
2. Copy the **full key** — it looks like `ak_live_<prefix>_<secret>` and is shown **exactly once**. Store it in your secret manager; you cannot retrieve it again (only re-generate).
3. Treat it like a password: never commit it, never log it. Rotate by generating a new key and revoking the old one (revoked keys return `401` immediately).

You now have two ways to send events. **The HTTP contract works today in any language**; the `AILogger` SDK is a thin convenience wrapper over it.

Path 1 — the AILogger SDK (recommended)

The SDK handles hashing, batching, retries, and idempotency for you. You pass the **raw** input/output; it computes the SHA-256 at source and discards the content.

> **Availability:** the TypeScript SDK is **live on npm** — `npm install @annexops/sdk`. The Python mirror (`annexops_sdk`) is coming to PyPI, and a **Go client** (module `github.com/annexops/runtime-logger-go`) is available on request — see the SDK README. Until a package lands for your language, it (and any other language) can use **Path 2 (HTTP)** below — it is the same wire contract and is fully supported.

TypeScript / Node (≥ 20)

```typescript
import { AILogger } from '@annexops/sdk';

const logger = new AILogger({
  apiKey: process.env.ANNEXOPS_API_KEY!,   // never hard-code
  systemKey: 'my-model-v1',                // a name you choose per AI system
});

// per inference — pass the RAW text; the SDK hashes it at source
logger.logInference({ input: prompt, output: response, modelVersion: 'gpt-4o',
        confidence: 0.98 });

// on shutdown, flush the buffer
await logger.close();
```

That is the whole integration. `logInference()` is non-blocking (it buffers locally in <1 ms); the SDK flushes automatically when the buffer fills or on an interval, and always on `close()`.

Python (stdlib only)

```python
from annexops_sdk import AILogger

logger = AILogger(api_key=os.environ["ANNEXOPS_API_KEY"], system_key="my-model-v1")

logger.log_inference(input=prompt, output=response, model_version="gpt-4o",
        confidence=0.98)

logger.close()
```

Go (≥ 1.21)

```go
import annexops "github.com/annexops/runtime-logger-go"

logger, _ := annexops.New(annexops.Config{
    APIKey:    os.Getenv("ANNEXOPS_API_KEY"), // never hard-code
    SystemKey: "my-model-v1",                 // a name you choose per AI system
})

// per inference — pass the RAW text; the SDK hashes it at source (SHA-256) and
        discards it
```

```
logger.LogInference(annexops.InferenceEvent{
    Input:        prompt,
    Output:       response,
    ModelVersion: "gpt-4o",
})

// on shutdown, flush the buffer
logger.Close(context.Background())
```

Same shape as the others: `LogInference` is non-blocking (it buffers locally); the SDK hashes at source, sends **hashes only**, flushes automatically when the buffer fills or on an interval, and `Close` flushes the remainder.

SDK configuration

Option	Default	Notes
`apiKey`	— (required)	Full `ak_live_…` key. Never logged or echoed in errors.
`baseUrl`	`https://annex-ops.vercel.app/api/v1`	Override only for a custom domain.
`systemKey`	—	Client-level default; can be overridden per call.
`flushAt`	`100`	Buffer size that triggers an automatic flush (hard-capped at 500).
`flushIntervalMs`	`5000`	Auto-flush cadence. `0` disables the timer.
`maxRetries`	`3`	Retries on network / 5xx / 429 with full-jitter backoff (500 ms → 30 s cap).
`requestTimeoutMs`	`10000`	Per-request timeout.

The SDK never retries a `401` or a `400` (those are permanent for that batch), and every retry re-sends the **identical** batch with the **same `event_ids`** — so retries are idempotent (the server de-duplicates).

Path 2 — direct HTTP (any language, works today)

The SDK is optional. Send events with a single authenticated POST.

Endpoint

```
POST https://annex-ops.vercel.app/api/v1/runtime-logs
```

Headers

Header	Value
`Authorization`	`Bearer ak_live_<your full key>`
`Content-Type`	`application/json`
`X-AnnexOps-Ingest-Version`	`1`

Body — a batch of 1–500 events, ≤ ~1 MB total:

```
{
  "events": [
    {
      "event_id": "b1f4…",          // your idempotency token, ≤100 chars (a UUID is
        ideal)
      "system_key": "my-model-v1",  // ≤200 chars; a name you choose per AI system
      "occurred_at": "2026-07-03T09:15:00Z",   // ISO-8601, your clock
      "input_hash": "<sha256 hex>",  // 64 lowercase hex chars
      "output_hash": "<sha256 hex>",
      "confidence": 0.98,            // optional, 0..1
      "model_version": "gpt-4o",     // optional, ≤100 chars
      "inference_ms": 420,           // optional, integer ≥0
      "human_override": false        // optional
    }
  ]
}
```

Only these fields are accepted (the schema is strict — any extra field rejects the event). `input_hash` / `output_hash` **must** be lowercase 64-character hex.

Response — always `202` on a structurally valid request:

```
{ "data": { "accepted": 1, "rejected": [] } }
```

`accepted` is the count queued for chaining. `rejected` lists any individually invalid events as `{ index, code, message }` — the rest of the batch still succeeds. A `202` with `accepted: 0` is **not** an error; it means every event in that batch was individually rejected (check `rejected`).

Worked example (bash)

```
KEY="ak_live_...your_full_key..."          # from your secret store — never commit
IN=$(printf  '%s' "what is 2+2?" | shasum -a 256 | cut -d' ' -f1)
OUT=$(printf '%s' "4"            | shasum -a 256 | cut -d' ' -f1)

curl -sS -X POST https://annex-ops.vercel.app/api/v1/runtime-logs \
  -H "Authorization: Bearer $KEY" \
  -H "Content-Type: application/json" \
  -H "X-AnnexOps-Ingest-Version: 1" \
  -d "{\"events\":[{
        \"event_id\":\"$(uuidgen)\",
        \"system_key\":\"my-model-v1\",
        \"occurred_at\":\"$(date -u +%Y-%m-%dT%H:%M:%SZ)\",
        \"input_hash\":\"$IN\",
        \"output_hash\":\"$OUT\",
        \"model_version\":\"gpt-4o\",
        \"confidence\":0.98
      }]}"
# → {"data":{"accepted":1,"rejected":[]}}
```

Compute `input_hash` / `output_hash` however your language hashes bytes — e.g. Python `hashlib.sha256(text.encode()).hexdigest()`, Node `crypto.createHash('sha256').update(text).digest('hex')`. Hash exactly the bytes you want to attest; there is no canonicalisation.

Retries & idempotency (if you skip the SDK)

- Retry on a network error, timeout, `5xx`, or `429` (honour `Retry-After` if present).
- **Never** retry a `401` or `400` — they will not succeed on retry.
- On retry, re-send the **identical** batch, including the same `event_id` values. The server de-duplicates on `(org, system_key, event_id)`, so a duplicate delivery is silently skipped — safe.

From events to your Article 12 record

Once events arrive:

1. **System-key mapping.** The first event under a new `system_key` appears in **Runtime Logs → System key mapping** as *Unassigned*. Map it to one of your registered AI systems. (Re-mapping affects the record's scope going forward; events already recorded keep their chain.)
2. **The hash chain.** Each event is appended to a tamper-evident chain for that `(org, system_key)`. The **chain-status banner** shows *intact* ✓ or flags the exact sequence where a break is detected.
3. **Verify.** The chain is re-verified server-side on demand — recomputing every row's hash — so you can prove integrity at any time.
4. **Export.** Click **Export Art 12 record** to download a signed JSON record for an AI system: its identity, the full event list, and the chain attestation.

Errors

HTTP	code	Meaning
`401`	`UNAUTHENTICATED`	Missing/malformed/revoked API key.
`400`	`VALIDATION_ERROR`	Malformed batch (bad JSON, empty `events`, >500 events, oversized body). Per-event issues appear in `rejected`, not here.

HTTP	code	Meaning
400	UNSUPPORTED_INGEST_VERSION	X-AnnexOps-Ingest-Version was set to anything other than 1.
500	INTERNAL_ERROR	Transient server error — safe to retry with backoff.

Every response carries an X-Correlation-ID (also in error.correlation_id) — include it if you contact support.

Limits & reference

	Value
Events per request	1–500
Request body	≤ ~1 MB
event_id	1–100 chars
system_key	1–200 chars
model_version	≤ 100 chars
input_hash / output_hash	^[0-9a-f]{64}$ (lowercase SHA-256 hex)
confidence	0.0–1.0
inference_ms	integer ≥ 0
Key format	ak_live_<prefix>_<secret>
Ingest version	1 (send X-AnnexOps-Ingest-Version: 1)

Security & privacy notes

- **Hashes only.** We never receive or store your prompts/responses — only their SHA-256 digests and the scalars you send.
- **Key handling.** The full API key is shown once at mint. Never commit it (the `ak_live_` prefix is recognised by secret scanners). Rotate by minting a new key and revoking the old.
- **Data residency.** All ingest and storage run in EU-Frankfurt.
- **Verification & export** happen in the AnnexOps portal (Vault → Runtime Logs), signed in — not via the SDK or your API key.

Questions? Your correlation-ID-stamped error responses and this guide should cover most integrations; reach out to your AnnexOps contact for anything else.