Siddes Observability Pack

Production debugging that is fast, privacy-first, and beginner-friendly.

Generated on 2026-01-25

Table of Contents

Siddes Observability Spec v1 (Privacy-first, Launch-grade)

File: docs/OBSERVABILITY_SPEC_V1.md (source: sd_600_observability_docs_datadog_apply_helper_20260125_053723.sh)

This spec is designed to make production debugging fast **without** logging user content or identities.

It assumes Siddes' current topology:

- **Frontend**: Vercel (Next.js)
- **Backend**: DigitalOcean App Platform (Django)
- **Media**: Cloudflare R2 + optional Cloudflare Worker on `/m/*`

Siddes already provides a strong baseline:

- Every `/api/*` response includes `X-Request-ID`.
- Backend emits structured JSON-line request logs (`event="api_request"`) for `/api/*`.
- Next `/api` proxy generates/forwards `x-request-id` and often injects `requestId` into JSON responses.

1) Goals (what we must answer in 5 minutes)

Errors

- Are 5xx errors spiking? Which endpoint(s)? Which service (Vercel vs Django vs Cloudflare Worker)?

Latency

- Is p95 latency spiking? Which endpoint(s)? Is it regional or global?

Auth failures

- Are 401/403 spikes happening (especially for write endpoints)? Did cookies/CSRF break?

Media failures

- Are `sign-upload` or `commit` failing? Are `/m/*` reads failing at edge?

Inbox failures

- Is send latency high? Is send failing? Is the failure in Next (Vercel) or Django?

2) Logging Standard (privacy-safe)

Siddes backend currently emits:

- `event`, `request_id`, `viewer`, `method`, `path`, `status`, `latency_ms`, `side`, (and sometimes `query`)

Required stored fields (canonical)

Store these fields for every request log event:

- `ts` (timestamp)
- `event` (normalized to `api.request`)
- `service` (`siddes-backend`, `siddes-next`, `siddes-edge`)
- `env` (`prod|staging`)
- `release` (deploy SHA/version)
- `request_id`
- `method`
- `route` (template, not raw IDs; e.g. `/api/post/:post_id/like`)
- `status`, `status_class` (`2xx|3xx|4xx|5xx`)
- `latency_ms`
- `side` (`public|friends|close|work|unknown`)
- `client_version` (frontend build/version)

PII rules: NEVER log

- Passwords, OTPs, session cookies, auth/refresh tokens, `Authorization` header
- Email, phone, username, real name
- Post bodies, comments, inbox/DM text, bios, search queries
- Signed URLs, media keys, raw IP addresses

Identity handling (strict)

- Do **not** index/store raw `viewer` values.
- Either **drop** `viewer` or convert to a non-reversible hash (preferred).

3) Correlation / Tracing Plan (MVP)

Siddes already has correlation through `request_id`:

- Client → Next (/api proxy) → Django
- `x-request-id` forwarded
- Django logs `request_id` and returns `X-Request-ID`

Support Code rule

- For API/action failures: use `request_id`

- For UI error boundaries: use Next `error.digest` (if/when you add client telemetry lane)

"Like didn't work" workflow: 1) Grab the `requestId` in the failing JSON or the `X-Request-ID` response header 2) Search logs by `request_id` 3) Read `route`, `status`, `latency_ms` 4) If no backend log exists → it failed before reaching Django (Vercel runtime / browser issue)

4) Metrics + Dashboards (log-derived)

Build log-based metrics from request logs:

- **Count**: requests by route/status/method/side
- **Distribution**: latency_ms by route

Dashboards: 1) **Golden Signals**: RPS, 5xx rate, p95 latency, top failing routes, top slow routes 2) **Endpoint Explorer**: route table (rps, p95/p99, 4xx/5xx), status breakdown, sample logs 3) **Auth & Session**: 401/403 trends, write-method 401 outside `/api/auth/*`, csrf/me checks 4) **Media Pipeline**: sign-upload + commit success/fail + p95; `/m/*` edge status breakdown 5) **Inbox Reliability**: send success/fail + p95; split by Next vs Django where applicable

5) Alerts (page vs notify)

Page

- Global 5xx ratio > 1% for 5 minutes (or "5xx count > threshold" if you're early stage)
- Core routes (auth/me, auth/csrf, feed, post create/like, media commit, inbox send) failing > 2% for 3 minutes
- Core p95 latency > 1500ms for 10 minutes
- Media commit failures > 5% for 10 minutes
- Health endpoints failing consecutively

Notify

- Write-method 401 spike outside `/api/auth/*` (CSRF/cookie drift)
- 429 spikes (rate limit)
- Region-specific spikes

Every alert payload must include:

- top affected routes
- example request_ids to search immediately

6) Privacy-safe telemetry controls

- **Retention**: raw logs ≤ 7 days (recommended)
- **Indexing**: index errors + slow requests; minimize indexing of 2xx
- **Sensitive Data Scanner / redaction**: hash viewer IDs; redact query strings
- **Access tiers**:
- Oncall/SRE: raw logs + request_id search
- Engineers: dashboards; time-bound raw access
- Support: no raw logs; uses support codes

7) Implementation (recommended: Datadog)

See `docs/OBSERVABILITY_CLICKOPS_DATADOG.md` for a copy/paste runbook to:

- forward DigitalOcean App Platform logs to Datadog
- drain Vercel logs to Datadog
- Logpush Cloudflare `/m/*` traffic logs to Datadog
- configure Sensitive Data Scanner to hash/redact
- build dashboards + alerts

Siddes Observability - ClickOps Implementation (Datadog)

File: docs/OBSERVABILITY_CLICKOPS_DATADOG.md (source: sd_600_observability_docs_datadog_apply_helper_20260125_053723.sh)

This is the “do exactly this” setup to centralize Siddes logs + build dashboards/alerts, **without changing Siddes code**.

You will connect:

- DigitalOcean App Platform → Datadog Logs
- Vercel Drains → Datadog Logs
- Cloudflare Logpush (/m/*) → Datadog Logs

Step 1 - Create Datadog keys (5 minutes)

1) In Datadog: **Organization Settings** → **API Keys**

- Create an API key named `siddes-ingest`

2) In Datadog: **Organization Settings** → **Application Keys**

- Create an App key named `siddes-admin` (needed for some config APIs; safe to keep private)

Step 2 - Forward backend logs (DigitalOcean App Platform → Datadog)

DigitalOcean App Platform supports forwarding logs to Datadog.

1) DigitalOcean Control Panel → **Apps** → select your Siddes backend app 2) Go to **Settings** 3) Find **Log Forwarding** → click **Edit** 4) Choose **Datadog** 5) Paste your Datadog API key and select the correct Datadog site (US/EU) 6) Select the backend component(s) to forward 7) Save

Definition of Done

- In Datadog Log Explorer, search: `@event:api_request`
- You should see logs with `request_id`, `path`, `status`, `latency_ms`

Step 3 - Drain Vercel logs (Vercel → Datadog)

Vercel Drains can send logs to Datadog. Drains are available on Pro/Enterprise.

1) Vercel Dashboard → **Team Settings** → **Drains** 2) Click **Add Drain** 3) Choose **Logs** 4) Destination:

- Prefer: **Datadog integration** (Marketplace) if available
- Otherwise: **Custom HTTP endpoint** pointing to your Datadog Logs intake endpoint

5) Format: **NDJSON** (recommended) 6) Environment: **Production** 7) Save

Definition of Done

- In Datadog Log Explorer, search for Vercel logs and confirm you see fields like:
- `requestId`, `statusCode`, `path`, `executionRegion`

Step 4 - Logpush Cloudflare `/m/*` edge logs (Cloudflare → Datadog)

Goal: see whether `/m/*` is returning 200/302/401/403/5xx and where.

1) Cloudflare Dashboard → **Logs** → **Logpush** 2) Create a Logpush job (zone-level for your app domain) 3) Destination: **Datadog** 4) Endpoint: use Datadog HTTP intake endpoint (v1 is fine) 5) API Key: your Datadog API key 6) Dataset: `http_requests` 7) Filters:

- include only requests where **path begins with** `/m/`

8) Tags (as destination params):

- `ddsource=cloudflare`
- `service=siddes-edge`
- `ddtags=app:siddes,env:prod`

9) Submit

Definition of Done

- In Datadog logs, search: `service:siddes-edge @http.request.uri.path:/m/*`
- You should see status codes and edge metadata.

Step 5 - Enable Sensitive Data Scanner (hash viewer + redact query)

Siddes backend logs include `viewer` (e.g. `me_123`) and may include `query`. We will **hash** viewer and **redact** query before it becomes searchable/indexed.

In Datadog: 1) **Organization Settings** → **Sensitive Data Scanner** 2) Create a **Scanning Group** for **Logs**

- Filter: `@event:api_request`
- (Optional) restrict to service `siddes-backend`

3) Add Rule: **Hash viewer**

- Scope: scan attribute `viewer`
- Regex pattern: `^me_[0-9]+$`
- Action: **Hash**
- Tag: `siddes_sensitive:viewer`

4) Add Rule: **Redact query**

- Scope: scan attribute `query`
- Regex pattern: `.+`
- Action: **Redact** with replacement text `[redacted_query]`
- Tag: `siddes_sensitive:query`

5) Save + wait a couple minutes

Definition of Done

- New logs show `viewer` as a hashed token (or redacted), not raw `me_123`.
- `query` shows `[redacted_query]` (or is empty).

Step 6 - Normalize routes (avoid cardinality explosion)

We want `route` templates like `/api/post/:post_id/like` instead of raw IDs.

In Datadog → **Logs** → **Configuration** → **Pipelines** Create pipelines that match paths and set a constant `route` attribute via **String Builder**.

Create these (minimum set):

A) Post routes

- Filter: `@event:api_request @path:/api/post/*/like`
- Set `route` = `/api/post/:post_id/like`
- Filter: `@event:api_request @path:/api/post/*/reply`
- Set `route` = `/api/post/:post_id/reply`
- Filter: `@event:api_request @path:/api/post/*/replies`
- Set `route` = `/api/post/:post_id/replies`
- Filter: `@event:api_request @path:/api/post/*/quote`
- Set `route` = `/api/post/:post_id/quote`
- Filter: `@event:api_request @path:/api/post/*`
- Set `route` = `/api/post/:post_id`

B) Inbox routes

- Filter: `@event:api_request @path:/api/inbox/thread/*`
- Set `route` = `/api/inbox/thread/:thread_id`

C) Media (static already)

- `/api/media/sign-upload`
- `/api/media/commit`

(These don't need templating; just set `route=%{path}` if you want.)

Definition of Done

- In logs, you can facet/group by `route` without seeing thousands of unique values.

Step 7 - Create log-based metrics

In Datadog Log Explorer: 1) Query: `@event:api_request`

- Click **Generate Metric**
- Metric name: `siddes.api.request.count`
- Type: count
- Tags: `route,method,status,side,service,env`

2) Query: `@event:api_request`

- Generate metric from attribute: `latency_ms`
- Metric name: `siddes.api.request.latency_ms`
- Type: distribution
- Tags: `route,method,side,service,env`

Step 8 - Dashboards (MVP)

Dashboard 1: Golden Signals

- RPS: `siddes.api.request.count` (timeseries)
- 5xx: query logs or metric filtered where status is 5xx
- p95 latency: `p95(siddes.api.request.latency_ms)` overall + by route
- Top failing routes: table grouped by `route`
- Top slow routes: table grouped by `route`

Dashboard 2: Media Pipeline

- sign-upload success/fail rate
- commit success/fail rate
- p95 latency for sign/commit
- Edge `/m/*` status breakdown (from Cloudflare logs)

Dashboard 3: Auth & Session

- 401/403 on `/api/auth/*`
- Write-method 401 outside `/api/auth/*`

Dashboard 4: Inbox Reliability

- send success/fail and p95 on `/api/inbox/thread/:thread_id`

Step 9 - Alerts (MVP)

Paging

- Backend 5xx count > 10 in 5 minutes (group by route)
- Media commit failures > 5 in 10 minutes
- Health checks failing (if you add synthetic checks)

Notify

- Write-method 401 spike outside auth (cookie/CSRF drift)
- p95 latency > 1500ms for 10 minutes on core routes

Alert payload must include

- top routes
- sample `request_id` values

Siddes - Datadog Dashboards & Monitors Pack (Copy/Paste)

File: docs/OBSERVABILITY_DATADOG_DASHBOARDS_ALERTS.md (source: sd_601_observability_datadog_dashboards_alerts_apply_helper_20260125_054406.sh)

You said you finished the Part 4 checks (logs are flowing, `@event:api_request` is visible, privacy rules are active). Now do **exactly** this to get production-debuggable dashboards + alerts.

This pack is designed to work **without changing Siddes code**.

0) One-time setup in Datadog (2 minutes)

Create facets (if not already)

In **Logs** → **Explorer**, find a recent `@event:api_request` log. For each attribute below, click it and choose **Create facet**:

- `route` (string) *(from your Datadog pipeline templating)*
- `path` (string)
- `method` (string)
- `status` (number)
- `side` (string)
- `latency_ms` (number)
- `request_id` (string)

> Facets let you group dashboards by route/status and drill down fast.

1) Create 2 log-based metrics (skip if already created)

Go to **Logs** → **Explorer** and run this query:

A) Requests count metric

Query:

```
@event:api_request
```

Click **Generate Metric**:

- Metric name: `siddes.api.request.count`
- Type: `count`
- Tags to keep (important): `route,method,status,side`
- (Optional if you have them): `service,env,release`

B) Latency distribution metric

Query:

```
@event:api_request
```

Click **Generate Metric**:

- Metric name: `siddes.api.request.latency_ms`
- Type: `distribution`
- Measure: attribute `latency_ms`
- Tags to keep: `route,method,side`
- (Optional): `service,env,release`

Why distributions: you can do `p95(...)` cleanly.

2) Dashboard Pack

Create a new dashboard for each section below.

Dashboard 1 - Siddes: Golden Signals (Prod)

Widget 1: RPS (backend)

Type: **Timeseries (Metric)** Query:

- `sum:siddes.api.request.count{*}.as_rate()`

Widget 2: 5xx count (backend)

Type: **Timeseries (Logs)** Log query:

```
@event:api_request @status:[500 TO 599]
```

Compute: `count`

Widget 3: Error rate % (5xx / all)

Type: **Timeseries (Logs) + Formula** Query A (errors):

```
@event:api_request @status:[500 TO 599]
```

Query B (total):

```
@event:api_request
```

Formula:

- `100 * A / B`

Widget 4: Latency p95 (overall)

Type: **Timeseries (Metric)** Query:

- `p95:siddes.api.request.latency_ms{*}`

Widget 5: Top failing routes (5xx)

Type: **Top List (Logs)** Query:

```
@event:api_request @status:[500 TO 599]
```

Group by: `route` Compute: `count` Top: 10

Widget 6: Top slow routes (p95 latency)

Type: **Table (Metric)** Query:

- `p95:siddes.api.request.latency_ms{*} by {route}`

Sort by: p95 desc Limit: 15

Dashboard 2 - Siddes: Endpoint Explorer

Widget 1: Route table (RPS, p95)

Type: **Table (Metric)** Queries:

- RPS: `sum:siddes.api.request.count{*}.as_rate() by {route}`
- p95: `p95:siddes.api.request.latency_ms{*} by {route}`

Tip: In table settings, show both columns and sort by p95 or RPS depending on incident.

Widget 2: Status breakdown for a selected route

Type: **Timeseries (Logs)** Query:

```
@event:api_request @route:$route
```

Group by: `status` Compute: `count`

> Add a dashboard variable named `$route` (type: facet `route`).

Widget 3: Sample logs for selected route (with request_id)

Type: **Log Stream** Query:

```
@event:api_request @route:$route
```

Columns to show:

- `request_id`
- `method`
- `status`
- `latency_ms`
- `side`

Dashboard 3 - Siddes: Auth & Session

Widget 1: 401/403 on auth endpoints

Type: **Timeseries (Logs)** Query:

```
@event:api_request @path:/api/auth/* @status:(401 OR 403)
```

Compute: `count` Group by: `status` (optional)

Widget 2: Write-method 401 outside auth (CSRF/cookie drift detector)

Type: **Timeseries (Logs)** Query:

```
@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE) -@path:/api/auth/*
```

Compute: `count`

Widget 3: Top routes causing write 401

Type: **Top List (Logs)** Query:

```
@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE) -@path:/api/auth/*
```

Group by: `route` Compute: `count` Top: 10

Dashboard 4 - Siddes: Media Pipeline

Widget 1: sign-upload success vs fail

Type: **Timeseries (Logs) + Formula** A (fail):

```
@event:api_request @path:"/api/media/sign-upload" @status:[400 TO 599]
```

B (total):

```
@event:api_request @path:"/api/media/sign-upload"
```

Formula:

- `100 * A / B` (fail %)

Widget 2: commit fail %

Type: **Timeseries (Logs) + Formula** A (fail):

```
@event:api_request @path:"/api/media/commit" @status:[400 TO 599]
```

B (total):

```
@event:api_request @path:"/api/media/commit"
```

Formula:

- `100 * A / B`

Widget 3: commit p95 latency

Type: **Timeseries (Metric)** Query:

- `p95:siddes.api.request.latency_ms{route:/api/media/commit}`

Widget 4: Edge `/m/*` status (Cloudflare)

Type: **Timeseries (Logs)** Query (assuming Logpush tags from the runbook):

```
service:siddes-edge @http.request.uri.path:/m/*
```

Group by: status (field name depends on your CF dataset; pick the status attribute you see) Compute: count

> If your CF logs don't have `service:siddes-edge`, search `ddsource:cloudflare` and adjust.

Dashboard 5 - Siddes: Inbox Reliability

Widget 1: Inbox send error count

Type: **Timeseries (Logs)** Query:

```
@event:api_request @route:"/api/inbox/thread/:thread_id" @status:[400 TO 599]
```

Compute: `count`

Widget 2: Inbox send p95 latency

Type: **Timeseries (Metric)** Query:

- `p95:siddes.api.request.latency_ms{route:/api/inbox/thread/:thread_id}`

Widget 3: Inbox send failures by status

Type: **Top List (Logs)** Query:

```
@event:api_request @route:"/api/inbox/thread/:thread_id" @status:[400 TO 599]
```

Group by: `status` Compute: `count` Top: 10

3) Monitor Pack (Alerts)

Create these monitors in **Monitors** → **New Monitor**.

Monitor 1 (PAGE): Backend 5xx spike

Type: **Log Monitor** Query:

```
logs("@event:api_request @status:[500 TO 599]").index("*").rollup("count").last("5m") > 20
```

Message (paste):

- "Backend 5xx spike. Check Golden Signals → Top failing routes. Pull sample `request_id` and search logs."

Monitor 2 (PAGE): Core media commit failures

Type: **Log Monitor** Query:

```
logs("@event:api_request @path:\"/api/media/commit\" @status:[400 TO
  599]").index("*").rollup("count").last("10m") > 10
```

Message:

- “Media commit failing. Check Media Pipeline dashboard. Likely R2/Worker/token-gate or upstream error.”

Monitor 3 (NOTIFY): Write 401 spike outside auth

Type: **Log Monitor** Query:

```
logs("@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE)
  -@path:/api/auth/*").index("*").rollup("count").last("10m") > 25
```

Message:

- “Write 401 spike outside auth. This often indicates cookie/CSRF/origin drift. Check Auth & Session dashboard.”

Monitor 4 (PAGE): Core route latency regression (p95)

Type: **Metric Monitor** Query:

```
p95(siddes.api.request.latency_ms{route:/api/feed}) > 1500
```

Evaluation: last 10 minutes Message:

- “Feed p95 latency high. Check Endpoint Explorer sorted by p95. Pull slow `request_ids`.”

> If your feed route differs, update the `route:` tag to match your `route` facet value.

Monitor 5 (NOTIFY): Rate limit spike (429)

Type: **Log Monitor** Query:

```
logs("@event:api_request @status:429").index("*").rollup("count").last("10m") > 50
```

Message:

- “429 spike. Check which routes are being throttled; tune limits if needed.”

Monitor 6 (PAGE): Edge `/m/*` 5xx (Cloudflare)

Type: **Log Monitor** Query (adjust status attribute name if needed):

```
logs("service:siddes-edge @http.request.uri.path:/m/* @http.response.status_code:[500 TO
  599]").index("*").rollup("count").last("10m") > 20
```

Message:

- “Edge /m 5xx spike. Check Media Pipeline dashboard and Cloudflare health.”

4) “How to use this” in a real incident (60 seconds)

1) Open **Golden Signals** 2) Click the top failing `route` 3) In logs, copy a `request_id` 4) Search `request_id:<id>` and read:

- `status`, `latency_ms`, `route`, `side`

5) Fix starts from **where** it fails:

- auth 401 writes → CSRF/cookie/origin config

- media commit fails → R2/Worker token gating / upstream
- edge /m 403 → token mismatch or media privacy gate

If you want a “one dashboard only” version (super minimal), use Dashboard 1 only + Monitors 1-3.

Siddes Observability Fire Drill (30 minutes, beginner-proof)

File: docs/OBSERVABILITY_FIRE_DRILL.md (source: sd_602_observability_fire_drill_and_next_pack_apply_helper_20260125_054811.sh)

Goal: prove you can go from **a user complaint → a request_id → the exact failing route/status/latency** in Datadog in under 2 minutes.

This drill is safe. It does **not** require code changes.

0) Pick targets (no guessing)

This drill needs two URLs:

- **WEB_BASE** (your Vercel app URL) - used to test Next logs (inbox stub + `/api/health`)
- **API_BASE** (your Django backend URL) - used to test backend `api_request` logs

If you don't know them, they are already in your go-live docs:

- `docs/GO_LIVE_DO_VERCEL_CLOUDFLARE_RUNBOOK.md` (look for `SD_INTERNAL_API_BASE=`)

1) Prove request_id correlation (Backend)

Run (replace the URL if needed):

```
curl -s -D - "$API_BASE/api/auth/csrf" -o /dev/null | sed -n 's/^X-Request-ID: //p'
```

Copy the request id.

In Datadog → Logs Explorer:

- Search: `@event:api_request @request_id:<PASTE_ID>`
- You should see exactly one log line.
- Verify fields exist: `path`, `status`, `latency_ms`, `method`

■ Done when you can search by request_id and find the backend log.

2) Prove route templating works (cardinality control)

Run (random id is fine; it can 404/401 - we only care about the route template):

```
curl -s -o /dev/null -w "%{http_code}
```

```
" "$API_BASE/api/post/aaaaaaaaaaaaaaaa/like"
```

In Datadog logs:

- Search: `@event:api_request @path:/api/post/*/like`
- Confirm your pipeline sets `route=/api/post/:post_id/like`

■ Done when `route` is templated (not a million unique paths).

3) Prove the “CSRF/cookie drift detector” (write 401 outside auth)

Send an unauthenticated write request to a known write endpoint:

```
curl -s -o /dev/null -w "%{http_code}
" -X POST "$API_BASE/api/post/aaaaaaaaaaaaaaaa/like" -H "content-type: application/json" -d
  '{}'
```

Expected: **401** (write guard)

In Datadog logs:

- Search: `@event:api_request @status:401 @method:POST -@path:/api/auth/*`
- You should see the request.

■ Done when the query above finds your test request.

4) Media pipeline sanity (backend)

Test sign-upload without auth:

```
curl -s -o /dev/null -w "%{http_code}
" -X POST "$API_BASE/api/media/sign-upload" -H "content-type: application/json" -d '{}'
```

Expected: **401** or **403** (depending on your auth rules)

In Datadog logs:

- Search: `@event:api_request @path:"/api/media/sign-upload"`

■ Done when you can see the request and status.

5) Prove Next (Vercel) logs exist for inbox stub

Run:

```
curl -s -o /dev/null -w "%{http_code}
" "$WEB_BASE/api/inbox/threads"
```

In Datadog logs:

- Search for the string: `/api/inbox/threads`
- Confirm you can find the log entry from Vercel drain.

■ Done when Datadog shows Vercel logs for inbox.

6) Test monitors without breaking prod

You cannot safely “force 5xx” in production. Instead, for each monitor:

- Click **Test Notifications** (Datadog UI)
- Confirm you receive the notification/pager

Optional safe test:

- Temporarily lower a threshold (for 5 minutes) so a single test request trips it, then revert.

■ Done when you can receive a test notification and you know where to find the route/request_id evidence.

7) You are now production-debuggable

If a user reports “X didn’t work,” you can: 1) get request_id (from header or response JSON) 2) search in logs 3) see route/status/latency 4) jump to dashboards for the wider picture

Siddes Observability - Vercel/Next Pack (Inbox Stub + /api/health)

File: docs/OBSERVABILITY_VERCEL_NEXT_PACK.md (source: sd_602_observability_fire_drill_and_next_pack_apply_helper_20260125_054811.sh)

Important repo fact:

- `docs/INBOX_BACKEND_CONTRACT.md` says inbox is currently a **backend_stub** using **Next.js route handlers**:
- `frontend/src/app/api/inbox/threads/route.ts`
- `frontend/src/app/api/inbox/thread/[id]/route.ts`

That means:

- Django `@event:api_request` logs will **NOT** include inbox traffic.
- Inbox observability must come from **Vercel drained logs**.

This guide makes inbox debuggable in Datadog **without code changes**.

1) Find a Vercel inbox log entry (proof)

In Datadog Logs Explorer, search for:

```
/api/inbox/threads
```

Open one result and look at the attributes panel. Identify these 4 fields (names vary depending on your drain format):

- request id (often `requestId` or `request_id`)
- path/url (often `path` or `url`)
- status code (often `statusCode` or `status`)
- duration in ms (often `durationMs` or `duration_ms`)

2) Create facets (once)

For each attribute you found above:

- Click the attribute → **Create facet**

Create facets for:

- `next_request_id` (whatever the request id attribute is)
- `next_path`
- `next_status`

- `next_duration_ms`

Tip: If the log already uses `path/status/duration`, reuse those; do not duplicate.

3) Create 2 Next log-based metrics

A) Next requests count

In Logs Explorer query:

```
/api/inbox/
```

Generate metric:

- Name: `siddes.next.request.count`
- Type: count
- Tags: include your path/route field and status field (whatever you have)

B) Next duration distribution

Same query:

```
/api/inbox/
```

Generate metric:

- Name: `siddes.next.request.duration_ms`
- Type: distribution
- Measure attribute: your duration-ms field

4) Build “Inbox (Next)” dashboard

Widget 1: Inbox RPS

Metric timeseries:

- `sum:siddes.next.request.count{*}.as_rate()`

Widget 2: Inbox errors

Log timeseries:

- query: `/api/inbox/ @statusCode:[500 TO 599]` (adjust to your status field)
- compute: count

Widget 3: Inbox p95 latency

Metric timeseries:

- `p95:siddes.next.request.duration_ms{*}`

Widget 4: Top failing inbox routes

Top list (logs):

- query: `/api/inbox/ @statusCode:[500 TO 599]` (adjust)
- group by: path field
- compute: count

5) Add 2 Inbox monitors (Next)

Monitor A (PAGE): Inbox 5xx spike (Next)

Log monitor:

- query: `/api/inbox/` filtered to 5xx (adjust field)
- threshold: > 10 in 5 minutes

Monitor B (NOTIFY): Inbox p95 high (Next)

Metric monitor:

- `p95(siddes.next.request.duration_ms{*}) > 1500` for 10 minutes

6) Correlation workflow (when inbox fails)

When the UI says “send failed”:

- Go to Datadog logs, search `/api/inbox/thread/`
- Use the Vercel request id field (often `requestId`) to find the exact invocation
- If you also see a backend `request_id` for other calls in the same session, you can correlate by time window

This closes the inbox observability gap without touching Django.

Siddes Support Code Workflow (No Code Changes)

File: docs/OBSERVABILITY_SUPPORT_CODE_WORKFLOW.md (source: sd_603_support_code_workflow_docs_apply_helper_20260125_055031.sh)

This workflow turns “it didn’t work” into **one searchable key** in Datadog within 60 seconds, without collecting user content.

Siddes already provides the key for backend calls:

- `X-Request-ID` response header (and backend logs include `request_id`)
- Next proxy often injects `requestId` into JSON responses

1) What counts as a “Support Code”

A) API Support Code = `request_id`

- The `request_id` is the fastest path to the exact server log line.
- It’s safe to share because it does **not** include user content.

B) UI Support Code = “error digest”

- Next error boundaries can show `error.digest` (if/when exposed in UI).
- For now, use `request_id` for almost everything.

2) The 60-second operator loop (the whole point)

1) Get a `request_id` (see Section 3) 2) In Datadog logs search:

- `@event:api_request @request_id:<PASTE>`

3) Read:

- `route` (or `path`), `status`, `latency_ms`, `side`

4) Decide quickly:

- 401/403 → auth/session/CSRF/config drift
- 429 → rate limiting
- 5xx → backend/downstream issue
- 2xx but user says it failed → client/UI/state issue (capture via screenshots + timing; client telemetry later)

3) How to get `request_id` (Beginner-proof)

Option A (Best): Web browser DevTools

1) Open Siddes in Chrome / Edge / Safari (desktop) 2) Open **DevTools** → **Network** 3) Reproduce the failure (click Like / Send / Upload) 4) Click the failing request in Network list 5) Find `X-Request-ID` in **Response Headers** 6) Copy it → that is the Support Code

If the response is JSON, also look for a `requestId` field in the response body.

Option B (iPhone/iPad): Safari Web Inspector (works, but a bit setup-y)

1) On iPhone: **Settings** → **Safari** → **Advanced** → **Web Inspector** → **ON** 2) Connect iPhone to a Mac via cable 3) On Mac: Safari → **Settings** → **Advanced** → **"Show Develop menu"** → **ON** 4) On Mac Safari menu: **Develop** → **<Your iPhone>** → **<Your Siddes tab>** 5) Use Network panel → copy `X-Request-ID`

Option C (No devtools): Use Datadog by time window

If you cannot extract `request_id`, you can still locate it:

1) Ask for:

- exact time (to the minute) + timezone
- what action they did (like / send / upload / login)
- which Side they were in (public/friends/close/work)

2) In Datadog logs:

- filter to `@event:api_request @status:[400 TO 599]`
- narrow by `@route:` (or `@path:`) based on the action (examples below)
- narrow time window (±2 minutes)

3) Pick the matching log line(s) and use `request_id` from there going forward.

Action → likely route/path filters

- Like: `@path:/api/post/*/like` or `@route:/api/post/:post_id/like`
- Post create: `@path:/api/post` (or your create endpoint)
- Media sign: `@path:"/api/media/sign-upload"`
- Media commit: `@path:"/api/media/commit"`
- Inbox send (Next stub): search Vercel logs for `/api/inbox/thread/` (see `docs/OBSERVABILITY_VERCEL_NEXT_PACK.md`)
- Auth failures: `@path:/api/auth/*`

4) What to do after you find the request_id

If status is 401/403

- Open **Auth & Session** dashboard
- Check "Write-method 401 outside /api/auth/*" (cookie/CSRF/origin drift detector)

- Check `/api/auth/csrf` and `/api/auth/me` synthetic status (if enabled)

If status is 5xx

- Open **Golden Signals** → Top failing routes
- Confirm if it’s one endpoint or many
- Pull 3 request_ids to see if it’s consistent (same route/status pattern)

If media is failing

- Open **Media Pipeline** dashboard
- If sign-upload ok but commit missing → likely browser PUT failed (invisible to backend)
- treat as client/network issue; collect time + action details and check edge logs

5) What NOT to ask users for (privacy)

Never request:

- screenshots of private messages/posts
- passwords/OTPs/tokens
- “send me your signed upload URL”
- email/phone unless strictly needed for account recovery (not debugging)

For debugging, you only need:

- time, action, side, and support code (request_id) if possible.

Siddes Support Response Macros (Copy/Paste)

File: docs/SUPPORT_RESPONSE_MACROS.md (source: sd_603_support_code_workflow_docs_apply_helper_20260125_055031.sh)

Use these messages to gather the minimum info without collecting private content.

A) First response (ask for Support Code + timing)

“Thanks - I can check the server logs. Please send: 1) The exact time it happened (and your timezone) 2) What you clicked (Like / Send / Upload / Login) 3) Which Side you were in (Public / Friends / Close / Work) 4) If you can: the **Support Code / Request ID** (it’s in the `X-Request-ID` header on the failing request).”

B) If they don’t know how to get Request ID

“No worries - if you can’t grab the Request ID, send the time (to the minute), what you clicked, and which Side you were in. I’ll locate it in logs.”

C) Confirm you found it (reassuring + specific)

“Got it. I found your request in logs:

- Route: <route>
- Status: <status>
- Request ID: <request_id>

We’re tracking this now.”

D) Known classes (choose one)

Auth/Session (401/403)

“This looks like an auth/session issue (401/403). We’re checking cookie/CSRF configuration and will push a fix.”

Server error (5xx)

“This is a server-side error (5xx). We’re investigating the failing endpoint and deploying a fix.”

Rate limit (429)

“This was rate-limited (429). We’re tuning limits so normal usage doesn’t get blocked.”

Media upload

“This looks like an upload pipeline issue. We’re checking the sign/commit steps and edge delivery.”

E) Closeout (with lightweight verification)

“We deployed a fix. Please retry the same action. If it still fails, send the new Request ID and the time.”

Siddes Incident Note Template (Copy/Paste)

File: docs/INCIDENT_TEMPLATE.md (source: sd_603_support_code_workflow_docs_apply_helper_20260125_055031.sh)

Summary

- Start time:
- End time:
- Severity:
- Impact (what users saw):
- Scope (routes / services):

Detection

- Alert/monitor name:
- First signal observed:
- Dashboard(s) used:

Evidence (no PII)

- Example request_ids:
- Top routes affected:
- Status patterns:
- Latency patterns:

Root cause

- What broke:
- Why it broke:
- What changed (deploy/config):

Fix

- What we did:
- When deployed:

Verification

- What checks passed (healthz/readyz/auth/me/media):

Prevention

- New monitor / threshold changes:

- Runbook updates:
- Follow-ups:

Siddes Oncall Home (Single Pane)

File: docs/ONCALL_HOME.md (source: sd_604_oncall_home_runbook_apply_helper_20260125_055726.sh)

This page is the “start here” for incidents. It assumes you have Datadog dashboards + monitors set up.

1) The one thing you ask for: Request ID

If a user reports “it didn’t work,” your first move is:

- get **Request ID** (`X-Request-ID` / `requestId`)

Then search:

- Datadog Logs → `@event:api_request @request_id:<PASTE>`

Docs:

- `docs/OBSERVABILITY_SUPPORT_CODE_WORKFLOW.md`

2) Open these dashboards (in this order)

A) Golden Signals (Prod)

Purpose: “Are we on fire?”

- RPS
- 5xx rate
- p95 latency
- top failing routes
- top slow routes

B) Endpoint Explorer

Purpose: “Which route is broken / slow?”

- route table (rps + p95)
- status breakdown
- sample logs with request_id

C) Auth & Session

Purpose: “Did cookies/CSRF break?”

- 401/403 on auth endpoints
- write 401 outside auth detector

D) Media Pipeline

Purpose: “Can users upload/view media?”

- sign-upload fail %
- commit fail %
- edge /m status breakdown

E) Inbox Reliability

Purpose: “Are messages sending?”

- send fail count
- send p95 latency
- status breakdown

3) When an alert fires (what to do immediately)

Alert: Backend 5xx spike

1) Golden Signals → Top failing routes 2) Click failing route → Endpoint Explorer 3) Pull 3 `request_ids` from logs 4) Confirm if it’s one route or many

Alert: Write 401 spike outside auth

1) Auth & Session dashboard 2) Check recent deploy/config change (origins/cookies/CSRF) 3) Verify `/api/auth/csrf` + `/api/auth/me` health

Alert: Media commit failing

1) Media Pipeline dashboard 2) Check sign-upload vs commit 3) Check edge `/m/*` status 4) If sign ok but commit missing: treat as client PUT/network (collect time/action; investigate edge + client signals)

Alert: Inbox 5xx (Next)

1) Inbox dashboard (Next) 2) Check Vercel logs for `/api/inbox/` 3) Confirm whether any backend calls are also failing

4) How to write an incident note (no PII)

Use:

- `docs/INCIDENT_TEMPLATE.md`

5) Fire drill (keep your skills sharp)

Run anytime:

- `./scripts/obs/fire_drill.sh`

Docs:

- `docs/OBSERVABILITY_FIRE_DRILL.md`

Siddes Oncall Quickstart (Beginner)

File: docs/ONCALL_QUICKSTART.md (source: sd_604_oncall_home_runbook_apply_helper_20260125_055726.sh)

If you only memorize one flow, memorize this:

1) User report → Request ID → Datadog search

1) Get `X-Request-ID` from the failing request (best) 2) Search Datadog logs: `@event:api_request @request_id:<id>` 3) Read `status`, `route`, `latency_ms`

2) Fast classification

- 401/403 → Auth/session/CSRF/cookie drift
- 429 → Rate limiting
- 5xx → Server/downstream
- 2xx but UI says it failed → Client/UI state mismatch

3) Where to click next

- 5xx spike → Golden Signals → Endpoint Explorer → sample request_ids
- auth drift → Auth & Session dashboard
- media → Media Pipeline dashboard + edge `/m/*`
- inbox → Inbox dashboard (Next logs)

4) Don't collect private content

Never ask users for:

- passwords, OTPs, tokens
- post/DM text
- signed URLs

Only ask for:

- time (to the minute) + timezone
- action they took
- which Side
- request id (support code)

Siddes Observability Master Checklist (Rebuild From Scratch)

File: docs/OBSERVABILITY_MASTER_CHECKLIST.md (source: sd_605_observability_master_checklist_apply_helper_20260125_060017.sh)

This checklist is designed for a beginner. Follow it top-to-bottom. It assumes **no code changes**.

Phase 0 - Ground truth (already in repo)

■ Backend emits JSON request logs for `/api/*` with `event="api_request"` and `request_id` ■ Backend returns `X-Request-ID` and exposes it to browsers via CORS ■ Next proxy forwards `x-request-id` and often injects `requestId` into JSON ■ Go-live docs define API base and health checks

Definition of Done:

- You can see `@event:api_request` in your log tool (Datadog).

Phase 1 - Centralize logs (single pane)

1.1 DigitalOcean App Platform → Datadog Logs

Do:

- Enable DO log forwarding to Datadog

DoD:

- Datadog Logs shows `@event:api_request`

1.2 Vercel → Datadog Logs (log drain)

Do:

- Enable Vercel log drains to Datadog

DoD:

- Datadog Logs includes Vercel runtime logs
- Searching `/api/inbox/threads` returns at least one entry after a request

1.3 Cloudflare Logpush (/m/*) → Datadog Logs

Do:

- Enable Cloudflare Logpush filtered to paths starting `/m/`

DoD:

- Datadog Logs shows edge logs for `/m/*`
- You can see status breakdown (200/302/401/403/5xx)

Phase 2 - Privacy controls (non-negotiable)

2.1 Hash or drop viewer identifiers

Do:

- Use Sensitive Data Scanner (or pipeline) to hash `viewer`
- Drop `dev_viewer`

DoD:

- New logs never show `viewer=me_123` (raw)
- `viewer` is hashed or removed

2.2 Redact or drop query

Do:

- Redact attribute `query` to `[redacted_query]` or drop it

DoD:

- New logs never include full query strings

2.3 Route templating (cardinality control)

Do:

- Create pipelines that set `route` templates for dynamic paths:
- `/api/post/:post_id/like`
- `/api/post/:post_id/reply`
- `/api/post/:post_id/replies`
- `/api/post/:post_id/quote`
- `/api/inbox/thread/:thread_id`
- `/m/:key` (edge side)

DoD:

- Grouping by `route` produces a small, stable set of values (not thousands)

Phase 3 - Metrics (log-based)

Create 2 metrics from logs: 1) `siddes.api.request.count` (count) 2) `siddes.api.request.latency_ms` (distribution on `latency_ms`)

DoD:

- You can graph RPS from `siddes.api.request.count`
- You can graph p95 latency from `siddes.api.request.latency_ms`

Phase 4 - Dashboards

Build these dashboards: 1) Golden Signals 2) Endpoint Explorer 3) Auth & Session 4) Media Pipeline 5) Inbox Reliability

DoD:

- In Golden Signals, you can identify top failing route in < 30 seconds
- In Endpoint Explorer, you can drill down and copy a request_id

Phase 5 - Alerts (Monitors)

Create these monitors:

- PAGE: backend 5xx spike (count threshold)
- PAGE: media commit failures
- NOTIFY: write-method 401 spike outside `/api/auth/*`
- PAGE: feed p95 latency > 1500ms
- NOTIFY: 429 spike
- PAGE: edge `/m/*` 5xx spike

DoD:

- "Test notification" works for each monitor
- Each alert message tells you which dashboard to open and what to look for

Phase 6 - Fire Drill (prove it works)

Run:

- `./scripts/obs/fire_drill.sh`

DoD:

- You can find each test request in Datadog with the suggested queries

- You can locate a request_id and read status/latency/route

Phase 7 - Support workflow (real-world usage)

Use:

- `docs/OBSERVABILITY_SUPPORT_CODE_WORKFLOW.md`
- `docs/SUPPORT_RESPONSE_MACROS.md`

DoD:

- For a user report, you can respond with:
- “I found your request in logs: route/status/request_id”

within 2 minutes

Phase 8 - Oncall start page

Pin:

- `docs/ONCALL_HOME.md`
- `docs/ONCALL_QUICKSTART.md`

DoD:

- In an incident, you know exactly where to click first.

If you completed Phases 1-8, Siddes is production-debuggable without violating privacy.

Observability (Production Debugging, Privacy-First)

File: docs/OBSERVABILITY.md (source: sd_614_observability_md_rewrite_apply_helper_20260125_075509.sh)

This is the **main entry point** for Siddes observability.

If you are a beginner, start here:

- Open `docs/OBS_PACK_INDEX.md` and follow the read order.
- Pin `docs/ONCALL_HOME.md` for real incidents.

What Siddes already gives you (baseline)

Backend (Django)

- Every `/api/*` response includes `X-Request-ID`.
- Requests to `/api/*` emit JSON-line logs via `siddes.api` logger:
- `event`, `request_id`, `viewer`, `method`, `path`, `status`, `latency_ms`, `side`
- Write guard exists in production to block unauthenticated writes (default allowlist is `/api/auth/*`).

Frontend (Next.js)

- Core routes include `error.tsx` boundaries to show calm failure + retry:
- Feed, Post detail, Sets, Invites, Inbox
- The `/api` proxy generates/forwards `x-request-id` and often injects `requestId` into JSON responses.

The 5-minute questions (the whole purpose)

In 5 minutes, oncall must be able to answer:

- Are 5xx errors spiking? Which endpoint(s)? Which service (Vercel vs Django vs Cloudflare edge)?
- Is p95 latency spiking? Which endpoint(s)?
- Are auth failures (401/403) spiking on write endpoints (cookie/CSRF/origin drift)?
- Are media failures happening (sign-upload/commit) and/or `/m/*` failing at the edge?
- Is inbox failing (note: inbox is currently a Next stub, so it’s Vercel logs, not Django logs)?

Privacy-first logging rules (non-negotiable)

Required stored fields (canonical)

Store only request metadata needed for debugging:

- `request_id`, `route` (templated), `method`, `status`, `latency_ms`, `side`

- plus operational tags: `service`, `env`, `release`, `client_version` (if available)

NEVER store (PII / secrets / content)

- passwords, OTPs, session cookies, auth/refresh tokens, `Authorization`
- email/phone/username/name
- post bodies, comments, inbox/DM text, bios, search queries
- signed media URLs, media keys, raw IP addresses

Mandatory ingestion transforms (no code changes)

1) **Hash or drop** `viewer` (do not index/store raw `me_123`) 2) **Drop/redact** `query` 3) Convert dynamic paths → **templated** `route` (cardinality control), e.g.:

- `/api/post/:post_id/like`
- `/api/inbox/thread/:thread_id`
- `/m/:key`

Policy docs:

- `docs/OBSERVABILITY_RETENTION_ACCESS_POLICY.md`
- `docs/OBSERVABILITY_PRIVACY_AUDIT_CHECKLIST.md`

Event taxonomy (minimal)

Server/edge:

- `api.request` (Django request logs; derived from `event="api_request"`)
- `next.request` (Vercel runtime logs for Next route handlers - required for Inbox)
- `media.edge_request` (Cloudflare `/m/*` logs)

Client (optional, log-only lane):

- `client.error.boundary`
- `client.net.request_failed`
- `client.media.put_failed`

Datadog setup + dashboards + alerts (copy/paste)

ClickOps implementation

- `docs/OBSERVABILITY_CLICKOPS_DATADOG.md`

Dashboards & monitors pack

- `docs/OBSERVABILITY_DATADOG_DASHBOARDS_ALERTS.md`

SLOs + burn-rate alerts (smarter paging)

- `docs/OBSERVABILITY_SLOS_V0.md`
- `docs/OBSERVABILITY_BURN_RATE_ALERTS.md`

How to debug a user report in 60 seconds (Support Code workflow)

Always ask for Request ID (`X-Request-ID` / `requestId`). Then search:

- Datadog Logs: `@event:api_request @request_id:<PASTE>`

Use:

- `docs/OBSERVABILITY_SUPPORT_CODE_WORKFLOW.md`
- `docs/SUPPORT_RESPONSE_MACROS.md`

Incident notes:

- `docs/INCIDENT_TEMPLATE.md`

Readiness gates + drills (make sure it works)

Before every prod deploy (do not deploy blind)

- `docs/OBSERVABILITY_LAUNCH_GATE.md`
- `scripts/go_live_observability_gate.sh`

Fire drill (prove request_id + route templating)

- `docs/OBSERVABILITY_FIRE_DRILL.md`
- `scripts/obs/fire_drill.sh`

Incident drills (practice real scenarios)

- `docs/OBSERVABILITY_INCIDENT_DRILLS.md`
- `scripts/obs/incident_drills.sh`

“What am I seeing?” and “What do I fix first?”

- `docs/OBSERVABILITY_KNOWN_FAILURE_SIGNATURES.md`
- `docs/OBSERVABILITY_FIX_PLAYBOOK_MAP.md`

Rebuild everything anytime

- `docs/OBSERVABILITY_MASTER_CHECKLIST.md`

If you only pin ONE doc for incidents:

- `docs/ONCALL_HOME.md`

Siddes Burn-Rate Alerts (Smarter paging, less noise)

File: docs/OBSERVABILITY_BURN_RATE_ALERTS.md (source: sd_608_burn_rate_alerts_apply_helper_20260125_061745.sh)

Burn-rate alerts page you when you're **spending error budget too fast**. This avoids constant paging on tiny spikes.

This v0 uses:

- Log monitors (5xx) as the "bad events"
- Metric monitors (p95 latency) as "user pain"
- Route templates (`route`) from your pipelines

1) Concepts (simple)

If your SLO is 99.5% over 7 days, your error budget is 0.5%. Burn rate means: "how fast am I consuming that 0.5% budget?"

We'll implement a practical approximation:

- **Fast burn**: big outage now → page quickly
- **Slow burn**: steady degradation → notify and fix before budget is gone

2) Availability burn alerts (5xx)

We use **two monitors per core route group**:

A) FAST BURN (PAGE)

Trigger when 5xx is high in a short window. Example (Feed):

- Window: 5 minutes
- Threshold: 5xx count > 20

Datadog Log Monitor query:

```
logs("@event:api_request @route:/api/feed @status:[500 TO 599]").rollup("count").last("5m") >
  20
```

B) SLOW BURN (NOTIFY)

Trigger when 5xx is elevated over a longer window. Example (Feed):

- Window: 60 minutes
- Threshold: 5xx count > 60

Datadog Log Monitor query:

```
logs("@event:api_request @route:/api/feed @status:[500 TO 599]").rollup("count").last("1h") >
  60
```

These are starter thresholds. After 2 weeks of traffic data, tune them by baseline.

3) Latency burn alerts (p95)

Latency doesn't consume "error budget", but users feel it like outages. Use the same pattern:

A) FAST LATENCY (PAGE)

Example (Feed):

```
p95(siddes.api.request.latency_ms{route:/api/feed}) > 2000
```

Evaluate: last 10 minutes

B) SLOW LATENCY (NOTIFY)

Example (Feed):

```
p95(siddes.api.request.latency_ms{route:/api/feed}) > 1500
```

Evaluate: last 60 minutes

4) Core route groups (copy/paste templates)

Feed

- route: `/api/feed`

FAST 5xx:

```
logs("@event:api_request @route:/api/feed @status:[500 TO 599]").rollup("count").last("5m") >
  20
```

SLOW 5xx:

```
logs("@event:api_request @route:/api/feed @status:[500 TO 599]").rollup("count").last("1h") >
  60
```

FAST p95:

```
p95(siddes.api.request.latency_ms{route:/api/feed}) > 2000
```

SLOW p95:

```
p95(siddes.api.request.latency_ms{route:/api/feed}) > 1500
```

Auth: csrf + me

If you have route tags for these, use them. Otherwise use path. CSRF 5xx:

```
logs("@event:api_request @path:\"/api/auth/csrf\" @status:[500 TO
  599]").rollup("count").last("5m") > 10
```

ME 5xx:

```
logs("@event:api_request @path:\"/api/auth/me\" @status:[500 TO
  599]").rollup("count").last("5m") > 10
```

Latency:

```
p95(siddes.api.request.latency_ms{route:/api/auth/me}) > 1000
```

Media: commit (critical)

FAST commit failures (PAGE):

```
logs("@event:api_request @path:\"/api/media/commit\" @status:[400 TO
  599]").rollup("count").last("10m") > 10
```

SLOW commit failures (NOTIFY):

```
logs("@event:api_request @path:\"/api/media/commit\" @status:[400 TO
  599]").rollup("count").last("1h") > 30
```

Latency:

```
p95(siddes.api.request.latency_ms{route:/api/media/commit}) > 2500
```

Auth drift detector (write 401 spike outside auth) - NOTIFY

```
logs("@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE)
  -@path:/api/auth/*").rollup("count").last("10m") > 25
```

This is not burn-rate, but it’s your best “configuration broke” early warning.

5) Inbox (Next stub) burn alerts (Vercel logs)

Inbox is currently Next route handlers in your repo, so use Next metrics:

- `siddes.next.request.count`
- `siddes.next.request.duration_ms`

FAST 5xx (PAGE) - adjust fields:

- Log monitor filtered to 5xx on `/api/inbox/` for 5m

FAST p95 (PAGE):

```
p95(siddes.next.request.duration_ms{*}) > 2000
```

SLOW p95 (NOTIFY):

```
p95(siddes.next.request.duration_ms{*}) > 1500
```

6) Alert messages (always include these)

Every monitor should say: 1) Which dashboard to open first (Golden Signals / Endpoint Explorer / Media / Auth) 2) What query to run 3) “Pull 3 request_ids and search them”

Example message: “Feed errors burning budget. Open Golden Signals → Endpoint Explorer. Search logs: `@event:api_request @route:/api/feed @status:[500 TO 599]` Pull 3 request_ids and inspect status/latency.”

After 2 weeks of traffic, we can switch from raw counts to true burn-rate % monitors.

Siddes - Fix Playbook Map (Signature → Evidence → Repo Knobs → Verify)

File: docs/OBSERVABILITY_FIX_PLAYBOOK_MAP.md (source: sd_613_fix_playbook_map_apply_helper_20260125_064746.sh)

This document links the most common production failure signatures to the **exact Siddes docs and config knobs** to check. It assumes **no application code changes** - only ClickOps/config and verification scripts.

If you have a `request_id`, always start here:

- Datadog Logs: `@event:api_request @request_id:<PASTE>`

0) Universal verification tools (use these first)

A) Proxy / health sanity (post deploy)

Run:

```
./scripts/post_deploy_smoke.sh <APP_ORIGIN> <API_ORIGIN> [MEDIA_TEST_URL]
```

Source: `scripts/post_deploy_smoke.sh` (tests `/healthz`, `/readyz`, `/api/health`, `/api/auth/me`, `/api/auth/csrf`, feed, optional media)

B) Observability Gate (before deploy)

Run:

```
WEB_BASE=<APP_ORIGIN> API_BASE=<API_ORIGIN> ./scripts/go_live_observability_gate.sh
```

Source: `docs/OBSERVABILITY_LAUNCH_GATE.md`

C) Observability Fire Drill (prove request_id + route templating)

Run:

```
WEB_BASE=<APP_ORIGIN> API_BASE=<API_ORIGIN> ./scripts/obs/fire_drill.sh
```

Source: `docs/OBSERVABILITY_FIRE_DRILL.md`

1) Signature: `backend_not_configured` / `proxy_fetch_failed` / no backend logs

What users see

- Login page shows `proxy_fetch_failed`
- API responses show `{ ok:false, error:"backend_not_configured" }`
- Inbox might work (Next stub), but real backend calls fail

Datadog evidence

- Vercel logs show errors for `/api/*`

- Django logs have **no matching** `api_request` for those calls

Where in Siddes docs

- `docs/FRONTEND_PROXY_DIAG.md`
- `docs/GO_LIVE_DO_VERCEL_CLOUDFLARE_RUNBOOK.md`
- `docs/GO_LIVE_CLICKOPS_DO_VERCEL_CLOUDFLARE.md`
- Template: `ops/deploy/vercel.env.example`

Config knobs (what to check)

On **Vercel**:

- `SD_INTERNAL_API_BASE=https://api.yourdomain.com`

On **backend (DigitalOcean)**:

- backend must be reachable and `/healthz` returns 200

Fix steps (ClickOps)

1) Vercel → Project → Settings → Environment Variables:

- set `SD_INTERNAL_API_BASE` exactly to your backend origin

2) Confirm backend health directly:

- `curl -i https://<backend>/healthz`

3) Confirm proxy health:

- `GET https://<app>/api/_diag` should show `ok:true` and `healthz.ok:true`

Verify

Run:

```
./scripts/post_deploy_smoke.sh https://<app> https://<api>
```

2) Signature: Write actions return 401/403 (CSRF/cookie/origin drift)

What users see

- “Can’t post”, “can’t like”, “send fails”
- Works in some browsers, not others
- Often begins right after a domain/proxy/CDN change

Datadog evidence

Write-method 401 spike outside auth:

```
@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE) -@path:/api/auth/*
```

Auth 401/403:

```
@event:api_request @path:/api/auth/* @status:(401 OR 403)
```

Where in Siddes docs

- `docs/PROD_DOMAINS_COOKIES_CSRF.md`
- `docs/GO_LIVE_DO_VERCEL_CLOUDFLARE_RUNBOOK.md`
- `docs/GO_LIVE_CLICKOPS_DO_VERCEL_CLOUDFLARE.md`
- Template: `ops/deploy/backend.env.prod.example`

Config knobs (what to check)

On **DigitalOcean (Django env vars)**:

- `DJANGO_ALLOWED_HOSTS=api.yourdomain.com`
- `DJANGO_CSRF_TRUSTED=https://app.yourdomain.com` *(must include Vercel/custom app origins)*
- Optional: `SIDDES_COOKIE_DOMAIN=.yourdomain.com` *(only if you want cross-subdomain cookies)*
- `DJANGO_DEBUG=0`

On **Vercel**:

- `SD_INTERNAL_API_BASE=https://api.yourdomain.com`

Fix steps (ClickOps)

1) Update DO env vars (allowed hosts + csrf trusted) 2) Redeploy backend 3) Confirm:

- `/api/auth/csrf` returns `{ok:true}`
- `/api/auth/me` returns valid JSON `{ok:true,...}` (even if viewer null when logged out)

4) If you use Cloudflare in front of your app domain:

- ensure it is not stripping cookies/headers

Verify

Run:

```
./scripts/post_deploy_smoke.sh https://<app> https://<api>
```

3) Signature: Media upload stuck (R2 CORS / PUT blocked) - sign-upload ok, commit missing

What users see

- Upload spinner never finishes

- Avatar upload fails
- Browser console shows CORS error (PUT blocked)

Datadog evidence

- You see sign-upload requests:
- `@event:api_request @path:"/api/media/sign-upload"`
- But commit is missing (or commit drops sharply):
- `@event:api_request @path:"/api/media/commit"`
- Often: sign-upload OK, then nothing (because browser PUT never reaches your servers)

Where in Siddes docs

- `docs/GO_LIVE_DO_VERCEL_CLOUDFLARE_RUNBOOK.md` (R2 bucket + CORS)
- `docs/GO_LIVE_CLICKOPS_DO_VERCEL_CLOUDFLARE.md` (R2 ClickOps)
- `docs/MEDIA_R2.md` (architecture)
- (Reference) overlay history: `sd_503_cloudflare_r2_cors_autopilot_*` (optional helper)

Config knobs (what to check)

On **Cloudflare R2 bucket CORS**:

- Methods: `PUT, GET, HEAD`
- Origins: your **exact** Vercel/custom app origins (no trailing slash)
- Headers: `content-type`

Fix steps (ClickOps)

1) Cloudflare Dashboard → R2 → Bucket → CORS 2) Set allowed origins exactly to your app origin(s) 3) Save

Verify

- Perform an upload in the app
- In Datadog:
- sign-upload happens
- commit happens
- Optional: run smoke + do a manual upload test

4) Signature: `/m/*` returns 401/403 for allowed users (Worker token secret mismatch)

What users see

- Post loads but image is broken

- Some viewers see media, others see 403
- Incognito gets blocked (expected), but logged-in viewer also blocked (bad)

Datadog evidence

- Backend sign/commit calls succeed
- Edge logs show `/m/*` returning 401/403 spikes
- Backend logs may show `GET /api/media/url?key=...` activity (if used)

Where in Siddes docs

- `docs/CLOUDFLARE_MEDIA_WORKER_TOKEN_GATE.md`
- `ops/deploy/cloudflare_worker_setup.md`
- `docs/MEDIA_R2.md`

Config knobs (what to check)

On **DigitalOcean (Django env vars)**:

- `SIDDES_MEDIA_TOKEN_SECRET=<secret>`
- Optional: `SIDDES_MEDIA_PRIVATE_TTL=600`

On **Cloudflare Worker secrets**:

- `MEDIA_TOKEN_SECRET` must match `SIDDES_MEDIA_TOKEN_SECRET`

Fix steps (ClickOps)

1) Confirm the DO secret value 2) In Worker project:

- `npx wrangler secret put MEDIA_TOKEN_SECRET`
- deploy Worker

3) Confirm Cloudflare route exists for `/m/*`

Verify

Follow `docs/CLOUDFLARE_MEDIA_WORKER_TOKEN_GATE.md` “What to test”

- Private-side media loads for allowed viewer
- Incognito fails (or fails after TTL)

5) Signature: `/m/*` returns 404 / bypasses worker / edge logs missing

What users see

- All media links 404

- Or media loads only in dev, not in prod

Datadog evidence

- No Cloudflare `/m/*` logs OR status is 404/5xx at edge
- Backend logs might show redirects in dev fallback, but production path not routed

Where in Siddes docs

- `docs/CLOUDFLARE_MEDIA_WORKER_TOKEN_GATE.md` (Route section)
- `ops/deploy/cloudflare_worker_setup.md`
- `docs/DOMAINS_AND_CLOUDFLARE_DNS_PLAN.md`

Config knobs (what to check)

In **Cloudflare**:

- Worker route: `https://yourdomain.com/m/*` (or `https://app.yourdomain.com/m/*` depending on your plan)

Fix steps (ClickOps)

1) Cloudflare → Workers & Pages → Routes 2) Add route for `/m/*` to the Worker 3) Ensure DNS for app domain points correctly

Verify

- Hit a real media URL:
- `curl -I https://app.yourdomain.com/m/<key>`
- expect 200/302 (depending on caching/token behavior)

6) Signature: 429 spikes (rate limiting too tight)

What users see

- “Try again” / intermittent failures
- More frequent for heavy usage or bursts

Datadog evidence

```
@event:api_request @status:429
```

Group by `route` to see where.

Where in Siddes docs

- `docs/THROTTLING.md` (scopes + env overrides)
- `docs/TIME_LIMIT_PLAYBOOK.md` (if timeouts also present)

Config knobs (what to check)

On **DigitalOcean (Django env vars)**, tune:

- `SIDDES_THROTTLE_INBOX_THREADS`
- `SIDDES_THROTTLE_INBOX_THREAD`
- `SIDDES_THROTTLE_INBOX_SEND`
- and other scope env vars listed in `docs/THROTTLING.md`

Fix steps (ClickOps)

1) Identify affected route(s) in Datadog 2) Find scope in `docs/THROTTLING.md` 3) Increase that scope env var 4) Redeploy backend

Verify

- 429 rate drops
- normal traffic no longer throttled

7) Signature: 5xx spike / outages

What users see

- “Server error”
- Feed doesn’t load

Datadog evidence

```
@event:api_request @status:[500 TO 599]
```

Golden Signals dashboard shows spike, Endpoint Explorer shows top failing routes.

Where in Siddes docs

- `docs/GO_LIVE_MASTER_RUNBOOK.md`
- `docs/DEPLOYMENT_GATES.md`
- `docs/GO_LIVE_ISSUE_REGISTER.md`

Fix steps (first moves)

1) Is it one route or many? 2) If many routes:

- suspect DB/infrastructure saturation
- restart/scale backend if needed

3) If one route:

- rollback the last deploy/config change

Verify

Run:

```
./scripts/post_deploy_smoke.sh https://<app> https://<api>
```

8) Signature: Inbox broken (Next/Vercel stub)

Repo fact:

- Inbox is currently served by Next route handlers:
- `frontend/src/app/api/inbox/threads/route.ts`
- `frontend/src/app/api/inbox/thread/[id]/route.ts`

What users see

- Inbox list doesn't load
- Send fails quickly

Datadog evidence

- Vercel logs show `/api/inbox/*` errors
- Django `@event:api_request` logs may show nothing for inbox

Where in Siddes docs

- `docs/INBOX_BACKEND_CONTRACT.md`
- `docs/INBOX_STUB_SMOKE_DEMO.md`
- `docs/INBOX_VISIBILITY_STUB.md`
- `docs/OBSERVABILITY_VERCEL_NEXT_PACK.md` (your observability pack)

Fix steps (ClickOps)

1) Confirm Vercel log drain is active (so you can see inbox logs) 2) Check Vercel deployment + env vars 3) Use the Next inbox dashboard/monitors from the Next Pack doc

Verify

Run:

```
curl -i https://<app>/api/inbox/threads
```

and confirm the request appears in Datadog.

9) Signature: "We're blind" (logs missing)

What you see

- No `@event:api_request` logs
- Dashboards empty

Where in Siddes docs

- `docs/OBSERVABILITY.md`
- `docs/OBSERVABILITY_CLICKOPS_DATADOG.md`
- `docs/OBSERVABILITY_LAUNCH_GATE.md`
- `docs/OBSERVABILITY_MASTER_CHECKLIST.md`

Fix steps (ClickOps)

1) Verify DO log forwarding 2) Verify Vercel drain 3) Verify Datadog index/pipelines still match `@event:api_request` 4) Re-run:

```
WEB_BASE=https://<app> API_BASE=https://<api> ./scripts/go_live_observability_gate.sh
```

If you want a final “everything in one page” version: use `docs/ONCALL_HOME.md` as your pin.

Siddes Observability Incident Drill Pack (Beginner, 45 minutes)

File: docs/OBSERVABILITY_INCIDENT_DRILLS.md (source: sd_611_observability_incident_drill_pack_apply_helper_20260125_062916.sh)

These drills make you confident you can handle real incidents. They are **safe**: they generate normal test traffic and expected error codes (401/404), and teach you where to click + what to search.

You do **not** need code changes.

Prereq:

- Logs are flowing to Datadog (`@event:api_request` works)
- Dashboards exist (Golden Signals, Endpoint Explorer, Auth, Media, Inbox)
- Fire drill already passed (`./scripts/obs/fire_drill.sh`)

Setup (2 minutes)

You need:

- `WEB_BASE` = your Vercel URL (e.g. https://siddes.app)
- `API_BASE` = your Django backend URL (e.g. https://api.siddes.app)

If you don’t know them, check your go-live runbook docs (look for `SD_INTERNAL_API_BASE=`).

Drill 1 - Auth Drift Detector (Write 401 spike outside auth)

Goal: prove you can detect “posting/liking broke due to cookie/CSRF/origin drift” quickly.

Step A: Generate the signal (safe)

Run an unauthenticated write request (expected 401 in prod):

```
curl -s -o /dev/null -w "%{http_code}
" -X POST "$API_BASE/api/post/aaaaaaaaaaaaaaaa/like" -H "content-type: application/json" -d
  '{}'
```

Step B: Verify the signal in Datadog (exact query)

In Datadog Logs Explorer search:

```
@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE) -@path:/api/auth/*
```

Step C: Where to click

- Dashboard: **Auth & Session**
- Look at “Write-method 401 outside auth”

- Look at “Top routes causing write 401”

Step D: What you would do in a real incident

- Confirm whether `/api/auth/csrf` and `/api/auth/me` are healthy
- If write 401 spikes suddenly after deploy → suspect:
- cookie domain mismatch
- CSRF trusted origins mismatch
- proxy/Cloudflare header/cookie stripping

Pass condition

- You can find the request in logs and see it in the Auth dashboard.

Drill 2 - Media Pipeline Break (sign/commit failing)

Goal: prove you can see sign/commit requests and interpret failures without looking at user content.

Step A: Generate sign-upload signal (safe)

Unauth sign-upload (expected 401/403):

```
curl -s -o /dev/null -w "%{http_code}
" -X POST "$API_BASE/api/media/sign-upload" -H "content-type: application/json" -d '{}'
```

Step B: Generate commit signal (safe)

Unauth commit (expected 401/403/400 depending on rules):

```
curl -s -o /dev/null -w "%{http_code}
" -X POST "$API_BASE/api/media/commit" -H "content-type: application/json" -d '{}'
```

Step C: Verify in Datadog

Search:

```
@event:api_request @path:"/api/media/sign-upload"
```

and:

```
@event:api_request @path:"/api/media/commit"
```

Step D: Where to click

- Dashboard: **Media Pipeline**
- sign-upload fail %
- commit fail %
- commit p95 latency
- edge `/m/*` status breakdown (if CF logs enabled)

How to reason (real incident)

- sign-upload fails → auth/config or backend issue
- sign-upload OK but commit missing → likely browser PUT to R2 failed (client/network), which backend can't see
- commit fails → auth rules, validation, upstream/R2 edge issues

Pass condition

- You can find both sign and commit requests in logs and see them on Media dashboard.

Drill 3 - Inbox Break (Next/Vercel stub observability)

Goal: prove you can debug inbox even if it never hits Django (because it's Next route handlers).

Step A: Generate inbox requests (safe)

```
curl -s -o /dev/null -w "%{http_code}
" "$WEB_BASE/api/inbox/threads"
curl -s -o /dev/null -w "%{http_code}
" "$WEB_BASE/api/inbox/thread/test"
```

Step B: Verify in Datadog (Vercel logs)

Search:

- `/api/inbox/threads`
- `/api/inbox/thread/`

If you created Next metrics per `docs/OBSERVABILITY_VERCEL_NEXT_PACK.md`, also verify:

- `siddes.next.request.count` exists
- `p95(siddes.next.request.duration_ms{*})` graphs

Step C: Where to click

- Inbox Reliability dashboard (Next)
- Vercel logs stream filtered to `/api/inbox/`

Pass condition

- You can find inbox logs in Datadog and see request durations/statuses.

After each drill: Practice the "support response"

Use:

- `docs/SUPPORT_RESPONSE_MACROS.md`

Write a pretend reply to a user using the request_id approach:

- “Found your request in logs: route/status/request_id.”

Bonus: Practice writing an incident note (5 minutes)

Use:

- `docs/INCIDENT_TEMPLATE.md`

Fill it with:

- start/end time (approx)
- what monitor would have fired
- 3 example request_ids from the drill

If any drill fails (what to fix)

- No backend logs → DO log forwarding / Datadog index/pipeline
- No Vercel inbox logs → Vercel drain
- No edge `/m/*` logs → Cloudflare Logpush filter/tagging
- `route` missing → route templating pipeline rules not applied
- privacy regression → Sensitive Data Scanner rules not matching

Siddes Known Failure Signatures (Datadog “What am I seeing?” Guide)

File: docs/OBSERVABILITY_KNOWN_FAILURE_SIGNATURES.md (source: sd_612_known_failure_signatures_apply_helper_20260125_064045.sh)

This is a beginner-friendly map from **symptom** → **Datadog evidence** → **first fix to try**. No code changes required.

Use this during incidents.

0) Always start with request_id (if you have it)

Datadog search:

- `@event:api_request @request_id:<id>`

Read:

- `route/path`, `status`, `latency_ms`, `side`

If you don’t have request_id:

- filter by time window (±2 minutes) + route + status.

1) CSRF / Cookie / Origin Drift (classic “everything broke after deploy”)

User symptom

- “I can’t like / post / send”
- “It keeps saying not authenticated”
- Works for some browsers, not others

Datadog signatures

A) Write-method 401 spike outside auth:

```
@event:api_request @status:401 @method:(POST OR PUT OR PATCH OR DELETE) -@path:/api/auth/*
```

B) Auth endpoints behaving oddly:

```
@event:api_request @path:/api/auth/* @status:(401 OR 403)
```

Dashboard to open

- **Auth & Session**

First fix to try (config)

- Verify cookie domain + CSRF trusted origins match current domains
- Verify Cloudflare isn’t stripping cookies/headers
- Verify `/api/auth/csrf` returns 200 and sets cookies correctly
- Re-run `scripts/go_live_observability_gate.sh`

2) Backend outage / downstream failure (true 5xx)

User symptom

- Feed doesn’t load
- Any action returns “server error”

Datadog signatures

A) Global 5xx spike:

```
@event:api_request @status:[500 TO 599]
```

B) Find top failing routes (group by route)

Dashboards

- Golden Signals → Endpoint Explorer

First fix to try

- Check if it’s one endpoint or many
- If many routes slow + 5xx: suspect DB / infra
- If one route: suspect a specific regression; roll back if needed

3) Media uploads broken (R2 / Worker / CORS / token gate)

User symptom

- “Upload stuck”
- “Image doesn’t appear”
- “Avatar upload fails”
- Media loads sometimes, sometimes 403

Datadog signatures

A) sign-upload fails:

```
@event:api_request @path:"/api/media/sign-upload" @status:[400 TO 599]
```

B) commit fails:

```
@event:api_request @path:"/api/media/commit" @status:[400 TO 599]
```

C) Edge /m failures (Cloudflare): Search logs:

- `service:siddes-edge @http.request.uri.path:/m/* @http.response.status_code:(401 OR 403 OR 5*)`

(adjust field names)

Dashboard

- Media Pipeline

Interpretation

- sign-upload fails → auth/config/back-end
- sign-upload OK but commit missing → browser PUT to R2 likely failed (client/network)
- commit fails with 4xx → validation/auth mismatch
- /m 403 spikes → token gate mismatch or side/privacy enforcement issue
- /m 5xx spikes → worker/R2 outage

First fix to try (config)

- Re-check R2 CORS allowed origins match **exactly** your web origin
- Confirm Worker env vars and token gate config
- Confirm media commit route has correct auth requirements

4) Inbox broken (Next/Vercel stub issues)

Repo fact:

- Inbox is currently served by Next route handlers (backend_stub). Django logs won't show it.

User symptom

- Inbox list doesn't load
- Send fails immediately

Datadog signatures

A) Vercel logs show /api/inbox/* errors

- Search: `/api/inbox/`

B) If you created metrics:

- `siddes.next.request.count` drop or spikes
- `p95(siddes.next.request.duration_ms{*})` spike

Dashboard

- Inbox Reliability (Next)

First fix to try

- Check Vercel deployment changes
- Check Vercel environment variables (API base / any feature flags)
- If inbox is expected to call backend, confirm proxy routes and CORS

5) Rate limiting (429)

User symptom

- “Sometimes it works, sometimes it says try again”
- Heavy users hit failures

Datadog signatures

```
@event:api_request @status:429
```

Group by `route` to see where it’s happening.

Dashboard

- Endpoint Explorer

First fix to try

- Increase limits for the affected route(s)
- Add targeted throttling rather than global

6) Performance regression (no errors, but slow)

User symptom

- App “spins” but eventually loads
- Works late at night but slow at peak

Datadog signatures

- p95 latency rising across multiple routes
- Top slow routes table shows broad slowdown

Dashboards

- Golden Signals (p95) → Endpoint Explorer (sort by p95)

First fix to try

- Look for deploy marker correlation
- If only one route: optimize/rollback that route
- If many routes: check DB connections/saturation

7) “It succeeded but user says it didn’t” (client state mismatch)

User symptom

- “Like didn’t change”
- “Message sent but didn’t appear”

Datadog signature

- Request returns 2xx quickly in logs, but UI didn’t update

What to do

- Ask for Request ID + time
- Confirm server returned 2xx
- Then treat as client cache/state issue:
- collect screenshots (without private content)
- check for client-side errors (once Part 2 client lane is enabled)

If you want the next item: a “Fix Playbook Map” that links each signature to the exact go-live docs and config knobs in your repo.

Siddes Launch Readiness - Observability Gate (Run Before Every Prod Deploy)

File: docs/OBSERVABILITY_LAUNCH_GATE.md (source: sd_610_observability_launch_gate_apply_helper_20260125_062727.sh)

This gate prevents “we deployed and now we’re blind” incidents. It requires **no code changes**.

Run this before each prod deploy and after any:

- Vercel drain changes
- Cloudflare logpush/worker changes
- backend logging/middleware changes
- Datadog pipeline/index changes

1) Required checks (pass/fail)

A) Backend logs present

Pass if:

- Datadog has recent `@event:api_request` logs in the last 10 minutes.

B) Request ID correlation works

Pass if:

- `curl $API_BASE/api/auth/csrf` returns an `X-Request-ID`
- searching Datadog by that request_id returns the log line

C) Privacy controls still active

Pass if:

- `viewer` is hashed or missing (not `me_123`)
- `query` is redacted or missing
- no auth secrets appear in logs (Authorization/Cookie/Bearer)

D) Route templating still active

Pass if:

- grouping by `route` shows a small stable set
- `/api/post/:post_id/like` templating still works

E) Vercel inbox logs present (Next stub)

Pass if:

- a request to `$WEB_BASE/api/inbox/threads` shows up in Datadog logs

F) Monitors are armed

Pass if:

- monitors exist for:
- backend 5xx spike
- media commit failures
- write 401 drift detector
- feed p95 latency
- edge /m 5xx (if CF logs enabled)

2) What to do if any check fails

1) Stop the deploy (or deploy only to staging) 2) Fix the blind spot:

- log forwarding / drain / logpush
- pipeline rules (hash/redact/route)

3) Re-run the gate 4) Only then deploy

3) Automated helper script

Use:

- `scripts/go_live_observability_gate.sh`

It runs safe curl checks and prints the exact Datadog queries to verify.

Siddes Observability Privacy Audit Checklist (Monthly)

File: docs/OBSERVABILITY_PRIVACY_AUDIT_CHECKLIST.md (source: sd_609_observability_privacy_audit_checklist_apply_helper_20260125_062008.sh)

This checklist ensures observability stays **privacy-safe** over time, especially as code and infrastructure change.

Do this once per month (or before a major launch).

A) Verify sensitive fields are not stored

A1) Viewer identifier is hashed or removed

In Datadog Logs Explorer:

- Query: `@event:api_request`
- Open a recent log line

Check:

- `viewer` is NOT present, OR it is hashed (not `me_123`)
- `dev_viewer` is NOT present

■ Pass criteria:

- No raw viewer IDs appear in any logs.

A2) Query strings are redacted/dropped

Check:

- `query` is either missing or equals `[redacted_query]`

■ Pass criteria:

- No user-entered query/search strings appear.

A3) No auth secrets appear

Search these patterns (each should return **zero** results):

- `Authorization:`
- `Bearer`
- `Set-Cookie:`
- `csrftoken=`
- `sessionid=`
- `refresh`

- `token`

■ Pass criteria:

- Zero hits in logs.

B) Verify cardinality controls (prevents accidental data leaks)

B1) Route templating is active

In logs, facet/group by `route`.

Check:

- You see a small stable set like:
- `/api/post/:post_id/like`
- `/api/inbox/thread/:thread_id`
- `/api/media/commit`
- etc.

■ Pass criteria:

- You do NOT see thousands of unique values.

B2) Raw `path` is not used for dashboards

Check your dashboards:

- They group by `route`, not raw `path`.

■ Pass criteria:

- Dashboards are stable and not exploding in unique paths.

C) Verify retention settings (privacy + cost)

C1) Raw logs retention

In Datadog:

- Logs → Configuration → Indexes (or Retention)

Check:

- Raw logs retention is set to ≤ **7 days** (recommended)
- Errors/sampled logs are retained appropriately, but not forever

■ Pass criteria:

- Raw logs do not retain beyond your policy.

C2) Telemetry DB retention (if enabled)

In backend env:

- `SIDDES_TELEMETRY_RETENTION_DAYS` is set (default 30)
- Purge job exists and ran recently

■ Pass criteria:

- Telemetry rows older than retention are gone.

D) Verify access controls (least privilege)

D1) Who can see raw logs?

Confirm:

- Support team does **not** have raw log access.
- Oncall/Engineers have access only as needed.

■ Pass criteria:

- Access aligns with “tiers” policy.

D2) Sensitive Data Scanner still enabled

Confirm:

- Scanner group still matches `@event:api_request`
- Hash rule for viewer is active
- Redact rule for query is active

■ Pass criteria:

- Rules are active and applied to new logs.

E) Verify the incident workflow stays privacy-safe

E1) Support Code workflow uses Request IDs only

Check:

- Support macros ask for time/action/side/request_id
- They do not ask for content or tokens

■ Pass criteria:

- No user content is requested.

E2) Incident template contains no PII

Check:

- Example request_ids only
- No user messages/post text

■ Pass criteria:

- Incident notes are clean.

F) Regression checks after deploy changes

Whenever you change:

- logging pipelines
- Cloudflare worker settings
- Vercel drains
- backend middleware

Run:

- `./scripts/obs/fire_drill.sh`

■ Pass criteria:

- You can still find logs by request_id and route templating still works.

If any check fails: 1) Disable indexing for the sensitive field immediately (stop the bleed) 2) Fix the pipeline rules (hash/redact) 3) Reduce retention temporarily 4) Re-run the fire drill

Siddes Observability Retention & Access Policy (v0)

File: docs/OBSERVABILITY_RETENTION_ACCESS_POLICY.md (source: sd_609_observability_privacy_audit_checklist_apply_helper_20260125_062008.sh)

This is a short, enforceable policy you can share with anyone who asks “what do we store?”

1) Data retained

A) Operational logs (Datadog)

- Purpose: production debugging
- Stored fields: request metadata only (route, status, latency, request_id)
- Identity: viewer is hashed or removed
- Content: never stored

Retention:

- Raw logs: **7 days** max (recommended)
- Aggregated metrics: 30-90 days

B) Telemetry events (DB)

- Purpose: counts-only product signals
- Stored fields: event name + viewer id + timestamp
- Retention: **30 days** (default)

C) Incident notes

- Purpose: learning + prevention
- Must not contain PII or user content
- Retention: as long as needed (safe because no PII)

2) Access tiers

Tier 1 - Oncall/SRE

- Can search raw logs by request_id
- Can view route/status/latency
- Cannot access secrets like HMAC salt

Tier 2 - Engineers

- Dashboards by default

- Raw logs time-bound when needed for incidents

Tier 3 - Support

- No raw logs
- Uses Support Code workflow (request_id/time/action/side)

3) Prohibited collection

Never collect or store in observability systems:

- tokens, cookies, Authorization headers
- usernames/emails/phones
- post bodies, comments, inbox messages
- signed media URLs or media keys
- raw IP addresses

4) Enforcement

- Sensitive Data Scanner rules must remain enabled
- Monthly privacy audit checklist must be completed

See:

- `docs/OBSERVABILITY_PRIVACY_AUDIT_CHECKLIST.md`

Siddes Observability SLOs v0 (No Code Changes)

File: docs/OBSERVABILITY_SLOS_V0.md (source: sd_607_slos_v0_apply_helper_20260125_061239.sh)

SLOs make alerts smarter and calmer. This v0 uses only what Siddes already logs:

- `@event:api_request` with `status` and `latency_ms`
- route templating (`route`) from your Datadog pipelines

This is **not** “enterprise SRE.” It’s a simple starter so you stop paging on noise.

1) Definitions (simple)

Availability SLI

For an endpoint group:

- **Good** = status is 2xx or 3xx
- **Bad** = status is 5xx
- (4xx usually means user/auth/validation; not “service down”)

Availability SLI formula:

- `good / total` where `total = good + bad`

Latency SLI

For an endpoint group:

- We track **p95 latency** and keep it below a threshold.

2) SLOs (starter targets)

These are intentionally forgiving for an early-stage product. Tighten later.

A) Feed SLO (core)

- Scope: `route:/api/feed`
- Availability: **99.5%** over 7 days (5xx only)
- Latency: p95 < **1500ms** over 30 minutes

B) Auth SLO (core)

- Scope: `/api/auth/me`, `/api/auth/csrf`
- Availability: **99.7%** over 7 days (5xx only)
- Latency: p95 < **800ms** over 30 minutes

C) Media SLO (core)

- Scope: `/api/media/sign-upload`, `/api/media/commit`
- Availability: **99.0%** over 7 days (5xx only)
- Latency: p95 < **2000ms** over 30 minutes
- Extra guardrail: `/api/media/commit` **4xx spike** monitor (can indicate auth/cors problems)

D) Inbox SLO (core user pain)

Because inbox is currently a Next stub in your repo, split it:

- **Inbox (Next)**: `/api/inbox/*` from Vercel logs/metrics
- **Inbox (Backend)**: only if/when those endpoints hit Django

Targets:

- Availability: **99.0%** over 7 days (5xx only)
- Latency: p95 < **1500ms** over 30 minutes

3) Error budget (what it means in plain English)

Example: 99.5% availability over 7 days:

- Total minutes in 7 days = 7 * 24 * 60 = 10080
- 0.5% budget = 50.4 minutes of “5xx time” allowed (across that endpoint group)

This is **not permission to be down** - it’s a tool to decide when to:

- stop shipping features
- focus on stability/perf

4) How to implement in Datadog (minimal)

4.1 Create a “5xx rate” monitor per SLO scope

For each scope, you can monitor:

- 5xx count
- or 5xx rate % (better once traffic is steady)

Example (Feed 5xx spike) - Log Monitor:

```
logs("@event:api_request @route:/api/feed @status:[500 TO 599]").rollup("count").last("5m") >
  10
```

Example (Feed latency regression) - Metric Monitor:

```
p95(siddes.api.request.latency_ms{route:/api/feed}) > 1500
```

4.2 Convert to SLOs later (optional)

Datadog has a dedicated SLO product that uses monitors as the SLI source. Once traffic is steady:

- define SLOs from these monitors
- track error budget burn

5) Tightening plan (what to do later)

After 2 weeks of data: 1) Look at typical p95 for each core route 2) Set latency threshold to ~2x typical p95 3) Lower availability error budgets gradually (99.0 → 99.5 → 99.7)

6) What not to do

- Don’t page on 4xx globally (it’s noisy)
- Don’t page on p95 for non-core endpoints
- Don’t group by raw path (use `route` templates only)

If you want, the next step is “burn-rate alerts” (fast + slow burn) for each SLO, still without code changes.

Siddes Observability Pack Index (Read Order)

File: docs/OBS_PACK_INDEX.md (source: sd_606_obs_pack_index_apply_helper_20260125_060359.sh)

If you are a beginner, read these in order.

1) What we are building (spec)

1) `docs/OBSERVABILITY_SPEC_V1.md`

- The rules: what we log, what we never log, what we must answer in 5 minutes

2) Set it up in the real world (ClickOps)

2) `docs/OBSERVABILITY_CLICKOPS_DATADOG.md`

- Connect DO → Datadog logs
- Connect Vercel → Datadog logs
- Connect Cloudflare `/m/*` → Datadog logs
- Hash/redact sensitive fields
- Route templating

3) Build dashboards + alerts (copy/paste)

3) `docs/OBSERVABILITY_DATADOG_DASHBOARDS_ALERTS.md`

- Create 2 log-based metrics
- Build dashboards (Golden Signals, Endpoint Explorer, Auth, Media, Inbox)
- Add monitors (5xx spike, media, auth drift, latency)

4) Prove it works (fire drill)

4) `docs/OBSERVABILITY_FIRE_DRILL.md` 5) `scripts/obs/fire_drill.sh`

5) Close the Next/Vercel inbox observability gap (important!)

6) `docs/OBSERVABILITY_VERCEL_NEXT_PACK.md`

- Inbox is a Next route-handler stub in your repo
- You must use Vercel drained logs to debug inbox

6) Support workflow (real incidents)

7) `docs/OBSERVABILITY_SUPPORT_CODE_WORKFLOW.md` 8) `docs/SUPPORT_RESPONSE_MACROS.md` 9) `docs/INCIDENT_TEMPLATE.md`

7) Oncall start page (pin this)

10) `docs/ONCALL_HOME.md` 11) `docs/ONCALL_QUICKSTART.md`

8) The “rebuild from scratch” checklist

12) `docs/OBSERVABILITY_MASTER_CHECKLIST.md`

Apply helpers (scripts you ran)

- `sd_600_...` → writes spec + clickops + playbook
- `sd_601_...` → writes Datadog dashboards/alerts doc
- `sd_602_...` → writes fire drill + Next inbox pack + helper script
- `sd_603_...` → writes support code workflow + macros + incident template
- `sd_604_...` → writes oncall home + quickstart
- `sd_605_...` → writes master checklist
- `sd_606_...` → writes this index

If you only pin ONE doc, pin:

- `docs/ONCALL_HOME.md`

Siddes Oncall Observability Playbook (Beginner-friendly)

File: docs/ONCALL_OBSERVABILITY_PLAYBOOK.md (source: sd_600_observability_docs_datadog_apply_helper_20260125_053723.sh)

This playbook assumes you have centralized logs + the Golden Signals dashboard.

0) The one thing you always ask for: Request ID

Siddes gives you `request_id` / `X-Request-ID` for `/api/*`.

When someone reports:

- “like didn’t work”
- “post failed”
- “upload stuck”

Your first move is: **get the Request ID** (from the response JSON `requestId` or the `X-Request-ID` header).

Then: 1) Search logs for that `request_id` 2) Read `route`, `status`, `latency_ms`

1) Triage flow (2 minutes)

1) Open **Golden Signals**

- 5xx spike? → jump to **Endpoint Explorer** and sort by errors
- p95 spike? → sort by slow endpoints

2) Check **Auth & Session**

- write 401 spike outside `/api/auth/*` often means CSRF/cookie/origin misconfig

3) Check **Media Pipeline**

- sign-upload ok but commit missing → likely browser PUT failed (client signal helps later)

4) Check **Edge /m/**

- If `/m/*` returns 401/403 unexpectedly → token gate or auth mismatch

2) Common incidents and what they look like

A) “Users can’t post / everything is 401”

Symptoms:

- write-method 401 spike outside `/api/auth/*`
- `/api/auth/csrf` failing or cookies not being set

What to do:

- confirm Vercel → API origin setup matches go-live runbook
- confirm Cloudflare proxy settings didn’t change cookie/headers

B) “Media uploads broken”

Symptoms:

- `/api/media/sign-upload` ok but `/api/media/commit` fails
- or commit success drops sharply
- edge `/m/*` status spikes

What to do:

- check R2 CORS settings (origin match exact)
- check Worker token-gate behavior and env vars

C) “Inbox send broken”

Symptoms:

- `/api/inbox/thread/:thread_id` 5xx/timeout spike
- or Vercel runtime errors if inbox is Next-side

What to do:

- check whether errors are in Vercel logs or Django logs (topology split)

3) How to answer a user report in 60 seconds

If you have the request_id:

- “I see your request hit the server at <time>, returned <status>, route <route>. We’re fixing <cause>.”

If you don’t have request_id:

- “Please send the Support Code shown on the error screen.” (use request_id/digest)

4) What “good” looks like (baseline)

- 5xx near zero
- p95 latency under ~1.5s on core routes

- media commit success stable
- auth 401 mostly limited to unauthenticated reads, not write actions