52 · Monitoring

Flutter Practice Notes

Contents

52 · Monitoring

Introduction

This module covers **observing a live Flutter app**: the **fundamentals** (monitoring vs observability, the three pillars — logs/metrics/traces), **crash reporting** (Crashlytics/Sentry, crash-free rate, symbolication, breadcrumbs), **analytics & custom metrics** (product events, funnels, privacy), and **performance monitoring & alerting** (startup/frames/network SLIs, dashboards, alerts, SLOs) — tied together into a **feedback loop** that turns production signals into action. It closes the delivery cycle after deployment (Module 51), consuming the logging you set up (Module 39) and enforcing the performance budgets you defined (Module 21/Module 47).

Why this module exists

Once an app is live on thousands of uncontrolled devices, **you can't see what's happening without instrumentation** — crashes, slow screens, dropped funnels, and errors are invisible until users complain (or leave). Monitoring makes production **observable**: it captures crashes with context, measures usage + performance, alerts you to regressions, and **closes the loop** so you fix and improve based on real data instead of guesses. It's what makes staged rollouts safe, launches sustainable, and the whole build→ship pipeline *learn*.

Module map

#	File	Topic	Level
1	01_monitoring_fundamentals.md	Monitoring vs observability; logs/metrics/traces; the feedback loop	🔴
2	02_crash_reporting.md	Crashlytics/Sentry, crash-free rate, symbolication, breadcrumbs	🔴
3	03_analytics_and_metrics.md	Product analytics, events, funnels, custom metrics, privacy	🟡
4	04_performance_monitoring_and_alerting.md	Perf monitoring (startup/frames/network), SLIs/SLOs, dashboards, alerts	🔴
5	05_monitoring_integration.md	Capstone: full observability + feedback loop	🔴

Cross-references: Logging (feeds monitoring): Module 39. Error handling (report failures): Module 38. Deployment/staged rollout (monitored): Module 51. Performance budgets: Module 21/Module 47. Firebase (Crashlytics/Analytics/Performance): Module 18. Security (no PII in telemetry): Module 37.

Prerequisites

39 Logging, 38 Error Handling, 51 Deployment, 21 Performance, 18 Firebase (Crashlytics/Analytics/Performance).

What you'll be able to do after this module

- Explain observability, the three pillars, and the monitoring feedback loop.
- Set up crash reporting with symbolication, breadcrumbs, and crash-free-rate tracking.
- Instrument product analytics + custom metrics (privacy-respecting).
- Monitor performance (startup/frames/network), define SLIs/SLOs, and alert on regressions.
- Assemble a full observability stack that closes the loop from production signals to action.

Capstone

Observability stack: For an app, wire crash reporting (Crashlytics/Sentry with symbolication + breadcrumbs + correlation ids), analytics (key events + a conversion funnel, PII-safe), and performance monitoring (startup/frame/network SLIs), with dashboards, alerts on regressions (crash-free rate / SLO breach), and a documented feedback loop (staged-rollout gate → triage → fix → iterate).

Summary

Monitoring makes a live app observable via logs/metrics/traces: crash reporting (with symbolication + context), analytics/custom metrics (privacy-safe), and performance monitoring with SLIs/SLOs + alerting. It closes the feedback loop — turning real production signals into safe rollouts, fast fixes, and data-driven iteration — completing the build→ship→observe→improve cycle.

Monitoring Fundamentals

> **Monitoring** = watching **known signals** to answer "is the app healthy?" (crash-free rate, error count, frame times). **Observability** = the property that lets you ask **new questions** about *why* — from rich, correlated telemetry — without shipping new code. Both are built on the **three pillars**: **logs** (discrete events — what happened), **metrics** (aggregated numbers over time — how much/how often), and **traces** (the path of one operation across the system — where time/failure went). The point isn't collecting data; it's the **feedback loop**: production signals → detect/diagnose → act (fix/rollback/iterate). Everything must be **PII-safe** and **not degrade the app**.

Introduction

This file establishes the conceptual frame: monitoring vs observability, the three pillars, and the feedback loop — before the crash/analytics/performance specifics. It clarifies terms teams conflate and sets the "signals → action" mindset.

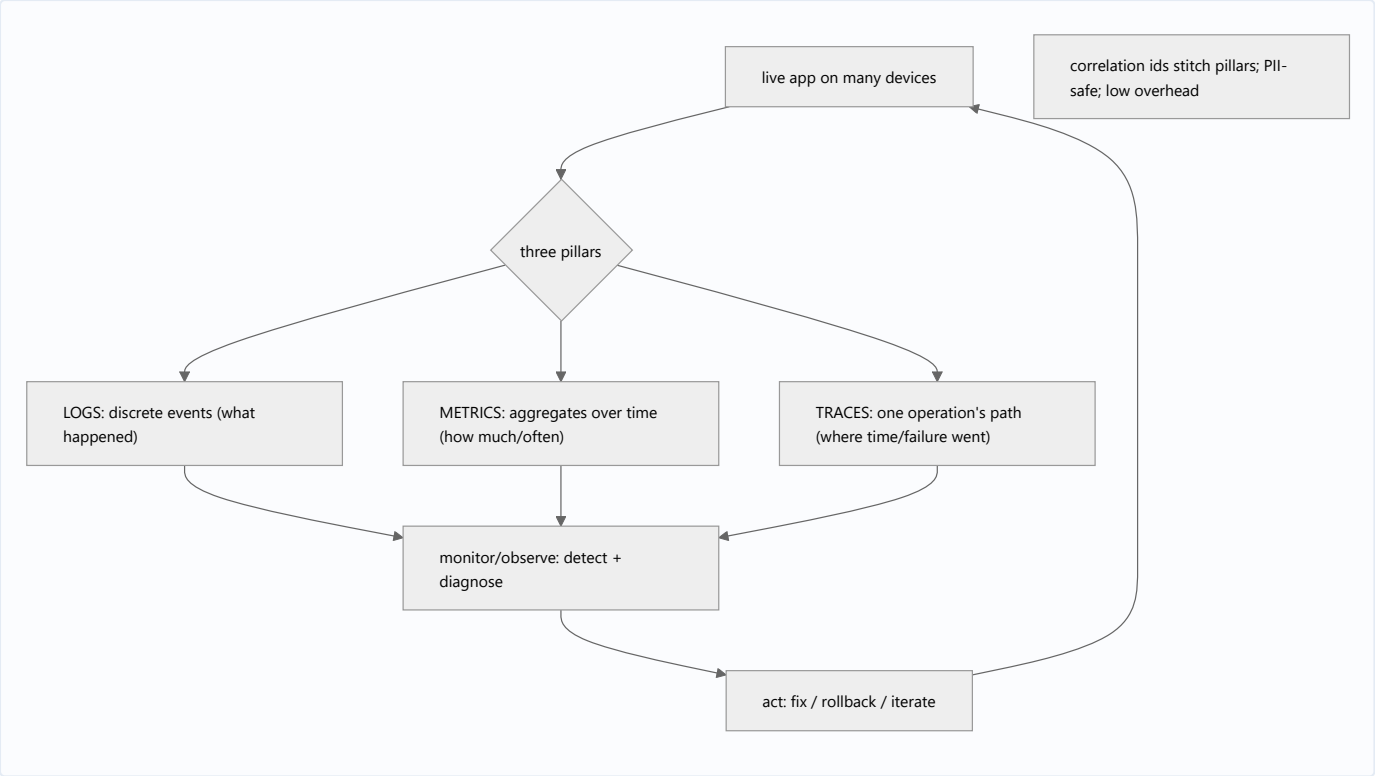
Why this concept exists

A live app runs on thousands of devices you can't see. Without instrumentation, you learn about crashes/slowness/drop-offs only from complaints — too late. Monitoring/observability give **visibility** so you can detect regressions fast (esp. during staged rollout), diagnose root causes, and improve based on **data, not guesses**. The three pillars organize *what* you collect; the feedback loop organizes *why*.

Real-world analogy

Monitoring is the **dashboard gauges** in a car — speed, fuel, temperature — showing known health at a glance. Observability is having **enough sensors + logs** that when something odd happens you can **investigate a new question** ("why did it overheat on hills?") without adding hardware. **Logs** = the trip journal (events), **metrics** = the odometer/averages (aggregates), **traces** = following one journey turn-by-turn (the path). The value is the **driver reacting** — slowing down, refueling, servicing — not the gauges themselves.

Internal Working



- **Monitoring vs observability**:
  - **Monitoring**: tracking **predefined signals/thresholds** to answer known questions — "is crash-free rate above 99.5%? are frames within budget?" Alerts fire on threshold breaches. Tells you **something is wrong**.
  - **Observability**: the system emits enough **rich, correlated** telemetry that you can **ask arbitrary new questions** and diagnose **why** — without deploying new instrumentation. Tells you **why it's wrong**.
  - You need both: monitoring to **detect**, observability to **diagnose**.
- **The three pillars**:
  - **Logs**: **discrete, timestamped events** (an error, a login, a request) — the granular "what happened," searchable/filterable (Module 39). Structured + leveled.
  - **Metrics**: **numeric measurements aggregated over time** (crash-free %, active users, p95 startup, frame-drop rate) — cheap to store, ideal for dashboards/alerts/trends.
  - **Traces**: the **end-to-end path of a single operation** across components (a request/user-action's spans + timings) — shows **where latency/failure occurred**; stitched by a **correlation/trace id** (Module 39).
  - Together they answer what (logs), how much (metrics), and where (traces) — and **correlate** via shared ids (a crash links to its logs/breadcrumbs and the request's trace).
- **The feedback loop (the actual point)**: **collect** (instrument) → **detect** (monitor/alert) → **diagnose** (observe: logs/traces/context) → **act** (fix/hotfix, halt staged rollout, iterate on

product) → back to collect. Monitoring without action is wasted; the loop is what makes it valuable (05_monitoring_integration.md).

- **What to observe (mobile)**: **stability** (crashes/ANRs — 02_crash_reporting.md), **performance** (startup/frames/network — 04_performance_monitoring_and_alerting.md), **product usage** (events/funnels — 03_analytics_and_metrics.md), and **errors** (handled failures). Attach **context** (app version, device, session) for slicing.
- **Constraints (non-negotiable)**:
  - **Privacy/PII-safe**: telemetry ships off-device to vendors — **never** log PII/secrets; redact + follow consent/regulation (Module 37/Module 39).
  - **Low overhead**: monitoring must **not** noticeably harm performance/battery/data — sample, batch, async (as with remote logging).
  - **Correlation**: use ids to stitch pillars per user action/session.
- **Tools (overview)**: **Firebase Crashlytics** (crashes) + **Analytics** (events) + **Performance Monitoring** (traces/metrics) — integrated, common for Flutter; **Sentry** (crashes + performance + logs, cross-platform); plus custom/backend telemetry. Choice covered in later files.

Memory Representation

Not runtime state you own — a **telemetry model**: logs (event stream), metrics (time-series), traces (span trees), correlated by ids + context (version/device/session). On-device, a small buffer batches telemetry for async upload; the analysis lives in the monitoring backend/dashboards.

Compiler Behavior

Not applicable (instrumentation is normal code + SDKs). Release builds are obfuscated → crash reports need **symbolication** with the archived mapping (Module 51/02_crash_reporting.md).

Runtime Behavior

Telemetry is captured during use and shipped **async/batched** (low overhead); the backend aggregates into metrics/dashboards and fires **alerts** on thresholds; correlation ids link a crash to its breadcrumbs/trace.

Flutter Engine Behavior

Performance monitoring hooks frame timings/startup from the engine (Module 21); crash SDKs capture native + Dart errors.

Dact VM Behavior

Uncaught Dart errors + isolate errors are captured by crash SDKs (Module 38); telemetry batching runs off the critical path.

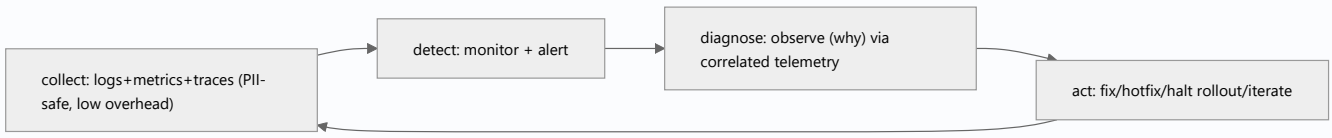
Examples

```
The three pillars answering different questions:
  LOGS:    "checkout_failed event at 12:03 for session X (NetworkFailure)"      -> what happened
  METRICS: "crash-free rate 98.7% (down from 99.6%); p95 startup 2.1s"          -> how much/trend
  TRACES:  "checkout took 3.4s: 3.1s in POST /order (server) + 0.3s render"     -> where the time went
  -> correlation id ties the crash -> its breadcrumbs (logs) -> the request trace

Monitoring vs observability:
  Monitoring:    alert when crash-free rate < 99.5% (KNOWN signal/threshold)
  Observability: after the alert, slice by version/device/screen to find WHY (NEW questions), no redeploy
```

```
Feedback loop:
  collect (instrument) -> detect (monitor/alert) -> diagnose (observe: logs/traces/context) -> act
(fix/rollback/iterate) -> repeat
  (monitoring WITHOUT the "act" step is wasted)
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Collecting data but never acting	Monitoring without the loop is waste	Close the loop: detect→diagnose→act
Confusing monitoring with observability	Different capabilities	Monitor to detect; observability to diagnose
Logging PII/secrets in telemetry	Privacy/compliance breach	Redact; PII-safe; consent (Module 37/39)
Heavy synchronous telemetry	Harms perf/battery/data	Async, batched, sampled
No correlation ids	Can't stitch crash↔logs↔trace	Correlation/trace ids + context
Only one pillar (e.g., only crashes)	Blind spots	Use logs + metrics + traces together
No alerting thresholds	Regressions unnoticed	Alert on key signals (crash-free/SLOs)
Unsymbolicated release crashes	Unreadable stacks	Symbolicate with archived mapping

Best Practices

- Use **monitoring to detect** (known signals/thresholds/alerts) and **observability to diagnose** (rich, correlated telemetry answering new questions); you need **both**.
- Instrument the **three pillars** (logs/metrics/traces) with **correlation ids + context** (version/device/session) so a crash links to its breadcrumbs and trace.
- **Close the feedback loop**: collect → detect → diagnose → **act** (fix/hotfix/halt rollout/iterate) — monitoring without action is wasted.
- Keep telemetry **PII-safe** (redact, consent) and **low-overhead** (async/batched/sampled); symbolicate release crashes with the archived mapping.

Performance

Monitoring must be **cheap**: async, batched, and sampled so it doesn't tax CPU/battery/data — the same discipline as remote logging (Module 39). Metrics are cheap to aggregate; traces/logs are sampled at volume. The payoff is catching real **performance/stability** regressions in production (Module 21).

Advantages / Disadvantages

- **+** Production visibility, fast detection + diagnosis, data-driven fixes/iteration, safe staged rollouts, closes the delivery loop.
- **−** Instrumentation + backend cost/setup, privacy discipline, overhead if done naively, alert tuning (noise vs signal), tool choice/integration.

Interview Questions

1. 🟢 **Monitoring vs observability?** — Monitoring watches known signals/thresholds to detect that something's wrong; observability is having rich correlated telemetry to ask new questions and diagnose why — without redeploying.
2. 🟢 **What are the three pillars?** — Logs (discrete events — what), metrics (aggregates over time — how much/trend), traces (one operation's path — where time/failure went).
3. 🟡 **How do the pillars work together?** — Correlation/trace ids + context stitch them: a metric alert leads to logs (events) and a trace (path) for the same action to find the cause.
4. 🟡 **What is the monitoring feedback loop, and why does it matter?** — Collect → detect → diagnose → act (fix/rollback/iterate); monitoring without acting on signals is wasted effort.
5. 🟡 **What constraints apply to mobile telemetry?** — PII-safe (redact/consent) and low-overhead (async/batched/sampled) so it doesn't leak data or harm performance/battery.
6. 🔴 **Why must release crashes be symbolicated, and how?** — Obfuscated release stacks are unreadable; symbolicate with the archived `--split-debug-info` mapping.
7. 🔴 **What should a mobile app observe?** — Stability (crashes/ANRs), performance (startup/frames/network), product usage (events/funnels), and handled errors — with slicing context (version/device/session).

Senior Engineer Tips

- Always design for the "act" step: pick signals + alerts that map to a decision (halt rollout, hotfix, prioritize a fix), or you're just collecting dashboards nobody uses.
- Thread a correlation id through logs → requests → crash context from day one; stitching crash ↔ breadcrumbs ↔ trace is the single biggest diagnosis multiplier.
- Keep telemetry async/batched/sampled and PII-free; monitoring that harms performance or leaks data is worse than none.

Architect Perspective

Monitoring/observability is the sensory system of a live app: the three pillars (logs/metrics/traces), correlated and contextual, let you **detect** (monitoring) and **diagnose** (observability), and the feedback loop turns those signals into action — safe rollouts, fast fixes, data-driven iteration. Built on the logging/error-handling foundation and feeding deployment's staged rollouts, it closes the build→ship→observe→improve cycle. The discipline is PII-safety + low overhead + acting on signals, not merely collecting them (02_crash_reporting.md, Module 39, Module 51).

Summary

- Monitoring detects (known signals/thresholds/alerts); observability diagnoses (rich correlated telemetry, new questions) — you need both.
- Three pillars: logs (events/what), metrics (aggregates/how much), traces (path/where) — correlated by ids + context.
- Close the feedback loop (collect→detect→diagnose→act); keep telemetry PII-safe + low-overhead; symbolicate release crashes.

Revision Notes

- Monitoring = watch known signals/thresholds (detect); observability = ask new questions from rich correlated telemetry (diagnose why) w/o redeploy. Need both.
- Pillars: logs (discrete events, what — Module 39), metrics (aggregates/time-series, how much/trend, alerts/dashboards), traces (one operation's spans, where latency/failure). Correlate via ids + context (version/device/session).
- Feedback loop: collect→detect→diagnose→act (fix/hotfix/halt rollout/iterate); PII-safe (redact/consent), low overhead (async/batched/sampled); symbolicate release crashes (archived mapping). Tools: Firebase Crashlytics/Analytics/Performance, Sentry.

Practice Questions

1. Distinguish monitoring from observability with an example.
2. What question does each of the three pillars answer?
3. Why is the "act" step essential, and what breaks without correlation ids?

Coding Questions

1. Classify a set of telemetry into logs/metrics/traces.
2. Show how a correlation id stitches a crash to its logs + a request trace.
3. Sketch the feedback loop for a crash-free-rate regression.

Mini Project

Observability model (Flutter/monitoring): For an app, define its observability model: which signals map to logs vs metrics vs traces, the correlation-id + context strategy (version/device/session), the key monitored signals + thresholds (e.g., crash-free rate, p95 startup), and the feedback loop (collect→detect→diagnose→act) — with PII-safety + low-

overhead constraints. Acceptance: three pillars applied with correct signal classification; correlation + context strategy; monitored signals/thresholds tied to actions; documented feedback loop; PII-safe + low-overhead; monitoring-vs-observability distinction clear.

Crash Reporting

> Crash reporting captures **uncaught crashes + fatal/non-fatal errors** from real devices — via **Crashlytics** or **Sentry** — and surfaces them as **grouped, symbolicated, contextualized** issues so you can find + fix what's actually breaking. The headline metric is **crash-free rate** (% of users/sessions without a crash — your stability SLI). To be useful, reports need **symbolication** (map obfuscated release stack traces back to source via the archived `--split-debug-info` mapping), **breadcrumbs** (the trail of events before the crash), **custom keys/context** (version/device/user-opaque-id/correlation id), and a **release-health** view (crashes per version) — all PII-safe.

Introduction

This file covers crash reporting concretely: wiring a crash SDK to global error handlers, symbolication, breadcrumbs + context, fatal vs non-fatal, crash-free rate + release health, and triage. It consumes the global error handling (Module 38) and the symbol mapping from deployment (Module 51).

Why this concept exists

Crashes on uncontrolled devices are invisible without reporting — and users rarely report them, they just churn. A crash reporter automatically captures every crash with the context needed to diagnose it, groups duplicates, tracks stability over time/versions, and alerts on spikes — the single most important production signal for app health.

Real-world analogy

A crash reporter is a **flight-data recorder + incident dashboard**: every crash "black box" is automatically retrieved with the **flight path leading up to it** (breadcrumbs), **decoded from raw codes to readable events** (symbolication), and **tagged with aircraft/route** (version/device/context). Identical incidents are **grouped**, and a **fleet-health board** shows crash-free rate per model (version) — so investigators fix the biggest, newest problems first instead of guessing from passenger complaints.

Internal Working

- **Capture uncaught errors**: route the **global handlers** to the crash SDK — `FlutterError.onError` (framework), `PlatformDispatcher.instance.onError` / `runZonedGuarded` (async/uncaught), and **native crashes** (SDK hooks) — so **every crash** is recorded (Module 38). Report **fatal** (crashes) and **non-fatal** (handled exceptions you still want visibility on) with stack + context.
- **Symbolication (essential for release)**: release builds are **obfuscated** (Module 51), so raw stacks are unreadable. Upload the **debug-info/symbol mapping** (`--split-debug-info` , dSYMs on iOS, mapping/ProGuard files on Android) to the crash service so it **maps stacks back to source**. Automate the upload in CI/CD; **without it, crashes are undiagnosable**.
- **Grouping/issues**: the service **groups identical crashes** into a single issue (by stack signature) with **occurrence count + affected users + versions + trend** — so you triage by **impact**, not noise.
- **Breadcrumbs + context** (diagnose the *why*):

    - **Breadcrumbs**: the **sequence of events** (navigations, taps, network results, logs) leading to the crash — fed from your logging facade (Module 39).
    - **Custom keys/context**: app **version/build**, **device/OS**, **session**, an **opaque user id** (not PII), the current **screen**, and a **correlation/trace id** to link to logs/traces.
    - Together they reconstruct the incident.
- **Crash-free rate (the key metric)**: **% of users (and % of sessions) that didn't crash** — your **stability SLI**. Track it per **release** (**release health**): a new version dropping crash-free rate is the signal to **halt a staged rollout** (04_performance_monitoring_and_alerting.md/Module 51).
- **Alerting**: alert on **crash-free-rate drops**, **new/regressed issues**, and **velocity spikes** (crashes/hour) — especially during rollout — so you react before wide exposure.
- **Triage workflow**: prioritize by **impact** (users × frequency × recency/newness), open the top issue → read **symbolicated stack + breadcrumbs + context** → reproduce/fix → verify crash-free rate recovers in the next release. Mark fixed issues to detect regressions.
- **ANRs / non-fatals**: Android **ANRs** (app-not-responding) and **OOMs** are tracked too; log important **handled** errors as non-fatals for visibility without crashing.
- **PII-safe** (Module 37/Module 39): **never** put PII/secrets in keys/breadcrumbs; use **opaque ids**; respect consent. Crash data is shipped to a vendor.
- **Tools**: **Firebase Crashlytics** (free, Flutter-integrated, crash-free rate + velocity alerts) vs **Sentry** (crashes + performance + errors, cross-platform, rich release health). Wire once, behind your error/logging layer.

Memory Representation

On-device: a small buffer of the current session's breadcrumbs + context, flushed with a crash report (async/on next launch). Backend: grouped issues (stack signature → occurrences/users/versions/trend), crash-free-rate time-series, and symbol mappings per version.

Compiler Behavior

Release obfuscation makes stacks opaque → the crash SDK relies on the uploaded **symbol mapping** (per build) to symbolicate; global handlers are top-level entry points capturing errors.

Runtime Behavior

On a crash, the SDK captures stack + breadcrumbs + context and reports it (often on **next launch**); non-fatals report immediately (async). The backend groups, symbolicates, updates crash-free rate, and fires alerts.

Flutter Engine Behavior

Framework errors flow via `FlutterError.onError`; native/engine crashes are caught by the SDK's platform hooks; ANRs/OOMs captured on Android.

Dart VM Behavior

Uncaught Dart errors (incl. isolates) route through the global handlers to the SDK; correlation ids from logging link Dart errors to their breadcrumbs/traces.

Examples

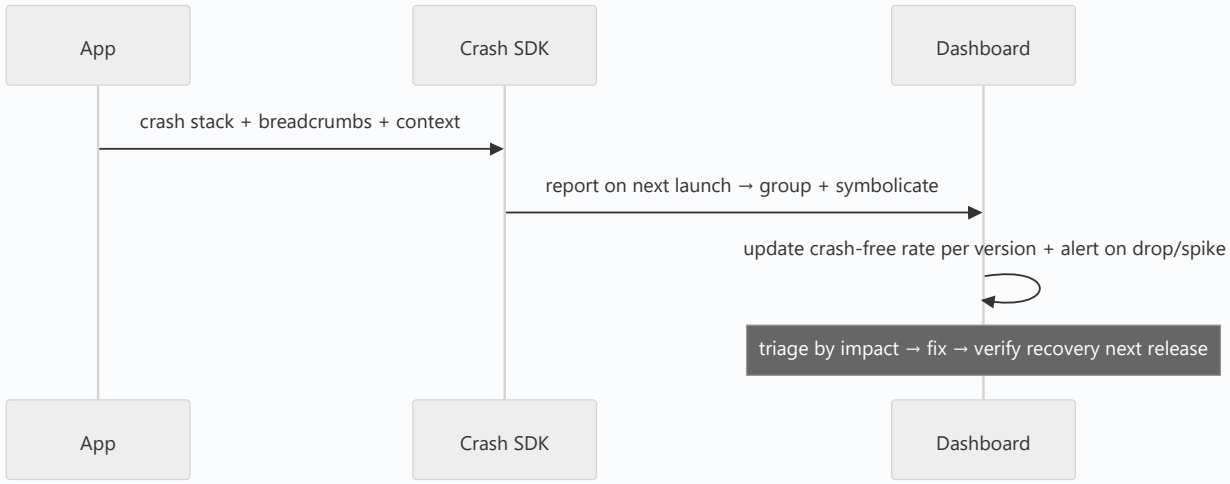
```
// Wire global handlers -> crash SDK (Crashlytics example) + non-fatals + context
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  FlutterError.onError = FirebaseCrashlytics.instance.recordFlutterFatalError;   // framework
  PlatformDispatcher.instance.onError = (e, s) {                                  // async/uncaught
    FirebaseCrashlytics.instance.recordError(e, s, fatal: true);
    return true;
  };
  // Context/custom keys (opaque id — NO PII) + correlation id
  FirebaseCrashlytics.instance.setCustomKey('screen', 'home');
  FirebaseCrashlytics.instance.setUserIdentifier(opaqueUserId);
  runApp(const App());
}

// Non-fatal (handled error you still want visibility on) + breadcrumb
try {
  await repo.load();
} catch (e, s) {
  FirebaseCrashlytics.instance.recordError(e, s, fatal: false);   // non-fatal
  logger.warn('load_failed', {'correlationId': corrId});          // breadcrumb (Module 39)
}
```

```
Symbolication (CI/CD) — upload symbols so release crashes are readable:
  flutter build ... --obfuscate --split-debug-info=build/symbols   # produce mapping
  # upload build/symbols + iOS dSYMs + Android mapping to Crashlytics/Sentry (automated in CI)
  # WITHOUT this: release stacks are obfuscated garbage.

Triage by impact:
  sort issues by (affected users x frequency x recency) -> fix biggest/newest first
  watch crash-free rate PER VERSION -> halt staged rollout if a new version drops it
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
No symbol upload for release	Stacks unreadable	Upload `--split-debug-info` /dSYM/mapping (CI)
Only capturing framework errors	Misses async/native crashes	Wire `PlatformDispatcher.onError` /zone + native
PII in keys/breadcrumbs	Privacy breach (vendor-shipped)	Opaque ids; redact; consent
Ignoring crash-free rate per version	Miss regressions during rollout	Track release health; halt on drop
Not using breadcrumbs/context	Can't diagnose the "why"	Feed breadcrumbs + custom keys + correlation id
Triaging by recency alone	Fix low-impact issues	Prioritize by users × frequency × newness
Not logging non-fatals	Blind to handled failures	Record important non-fatals
No alerting on spikes	React too late	Alert on crash-free drop / velocity spike

Best Practices

- Route **all** global handlers (`FlutterError.onError` + `PlatformDispatcher.onError` /zone + native) to the crash SDK; report **fatal + important non-fatal** errors with stack + context.
- **Upload symbols** (`--split-debug-info` /dSYM/mapping) in CI so **release crashes symbolicate**; without it they're undiagnosable.
- Attach **breadcrumbs + custom keys + correlation id + version/device/session** (PII-safe, opaque ids) to diagnose the *why*; feed breadcrumbs from your logging facade.
- Track **crash-free rate per version** (release health), **alert** on drops/spikes (halt staged rollout), and **triage by impact** (users × frequency × recency).

Performance

Crash reporting is low-overhead: reports are batched/async (often sent on next launch), breadcrumbs are a small ring buffer, and non-fatals are cheap. It must not degrade the app — the SDKs are designed for this. The value is catching stability regressions early (esp. during rollout).

Advantages / Disadvantages

- **+** Automatic capture of all crashes with context, grouping, symbolicated stacks, crash-free-rate/release-health tracking, spike alerts, impact-based triage.
- **−** Symbol-upload setup, PII discipline, vendor dependency, alert tuning, non-fatal noise if over-reported.

Interview Questions

1. 🟢 **What is crash-free rate, and why does it matter?** — The % of users/sessions without a crash — the key stability SLI; tracked per version to detect regressions and gate staged rollouts.
2. 🟢 **Why is symbolication required, and how is it done?** — Release builds are obfuscated (unreadable stacks); you upload the `--split-debug-info` /dSYM/mapping so the service maps stacks back to source.
3. 🟡 **How do you capture all crashes in Flutter?** — Route `FlutterError.onError` (framework) + `PlatformDispatcher.onError` /zone (async/uncaught) + native SDK hooks to the crash reporter; also record important non-fatals.
4. 🟡 **What context makes a crash diagnosable?** — Breadcrumbs (event trail), custom keys (screen/version/device), an opaque user id, and a correlation id linking to logs/traces — all PII-safe.

5. 🟡 **How do you triage crashes?** — By impact (affected users × frequency × recency/newness) — fix the biggest, newest issues first using the symbolicated stack + breadcrumbs.
6. 🔴 **How does crash reporting integrate with staged rollout?** — Watch crash-free rate per version during rollout; a drop (or a new-issue spike) triggers halting/rolling back the rollout (Module 51).
7. 🔴 **Fatal vs non-fatal reporting — why both?** — Fatal = crashes (stability); non-fatal = handled errors you still want visibility on (early warning) without crashing the app.

Senior Engineer Tips

- Automate symbol uploads in CI on every release; the day a crash spikes, unsymbolicated stacks turn a 10-minute fix into an all-day guess.
- Wire crash reporting behind your error/logging layer so breadcrumbs + correlation ids come for free, and never let PII into keys/breadcrumbs — crash data leaves the device.
- Gate staged rollouts on crash-free rate per version and triage by impact; a new version dropping crash-free rate is your primary halt signal.

Architect Perspective

Crash reporting is the stability sensor of the observability stack: it turns invisible field crashes into grouped, symbolicated, contextualized issues and a crash-free-rate SLI that gates rollouts and drives triage. Built on the global error handlers + logging breadcrumbs + the deployment symbol mapping, and kept PII-safe/low-overhead, it's the single highest-value production signal — feeding the alerting/feedback loop and the safe-release process (Module 38, Module 39, Module 51, 05_monitoring_integration.md).

Summary

- Route all global handlers (+ native) to a crash SDK (Crashlytics/Sentry); report fatal + non-fatal with stack + context; upload symbols so release crashes symbolicate.
- Diagnose via breadcrumbs + custom keys + correlation id + version/device (PII-safe); crashes are grouped into issues.
- Crash-free rate per version = stability SLI; alert on drops/spikes (halt rollout); triage by impact (users × frequency × recency).

Revision Notes

- Capture: `FlutterError.onError` (framework) + `PlatformDispatcher.onError`/zone (async) + native + non-fatals → crash SDK (Crashlytics/Sentry). Fatal + non-fatal.

- Symbolicate: upload `--split-debug-info`/dSYM/Android mapping (CI) — else release stacks unreadable. Group duplicates into issues (count/users/versions/trend).
- Context: breadcrumbs (from logging) + custom keys (screen/version/device) + opaque user id + correlation id; PII-safe. Crash-free rate per version (release health) = stability SLI → alert on drop/spike → halt staged rollout; triage by impact (users×frequency×recency).

Practice Questions

1. Why must you upload symbols, and what breaks without it?
2. What context turns a crash into a diagnosable issue?
3. How does crash-free rate gate a staged rollout?

Coding Questions

1. Wire `FlutterError.onError` + `PlatformDispatcher.onError` to a crash SDK with context.
2. Record a non-fatal error + a breadcrumb with a correlation id.
3. Add CI symbol upload for release crash symbolication.

Mini Project

Crash reporting setup (Flutter/monitoring): Wire a crash SDK (Crashlytics/Sentry) to all global handlers (framework + async + native) reporting fatal + important non-fatal errors with breadcrumbs (from your logging facade) + custom keys + correlation id + version/device (PII-safe, opaque ids), automate symbol upload in CI for release symbolication, and set up crash-free-rate-per-version tracking + an alert (drop/spike) that gates staged rollout. Acceptance: all crash sources captured + symbolicated (CI symbol upload); breadcrumbs + context (no PII); grouped issues; crash-free-rate release health + alert → rollout halt; triage-by-impact plan.

Analytics & Custom Metrics

> Analytics answers **"what are users actually doing?"** — you log **events** (`screen_view`, `add_to_cart`, `purchase`) with **parameters + user properties**, then analyze **funnels** (step-by-step conversion), **retention**, and **custom metrics** to drive **product decisions**. The discipline: define a deliberate **event taxonomy** (consistent names/params — not ad hoc), instrument **key user actions + business outcomes** (not everything), respect **privacy/consent** (no PII, honor ATT/GDPR, opt-out), and keep it behind an **abstraction** (an `Analytics` interface) so it's swappable and testable. Analytics is about **learning + deciding**, distinct from crash/perf monitoring (health).

Introduction

This file covers product analytics: event taxonomy, funnels/retention/custom metrics, privacy/consent, and wrapping analytics behind an interface. It's the "what/why users do" pillar, complementing stability (crashes) and performance monitoring.

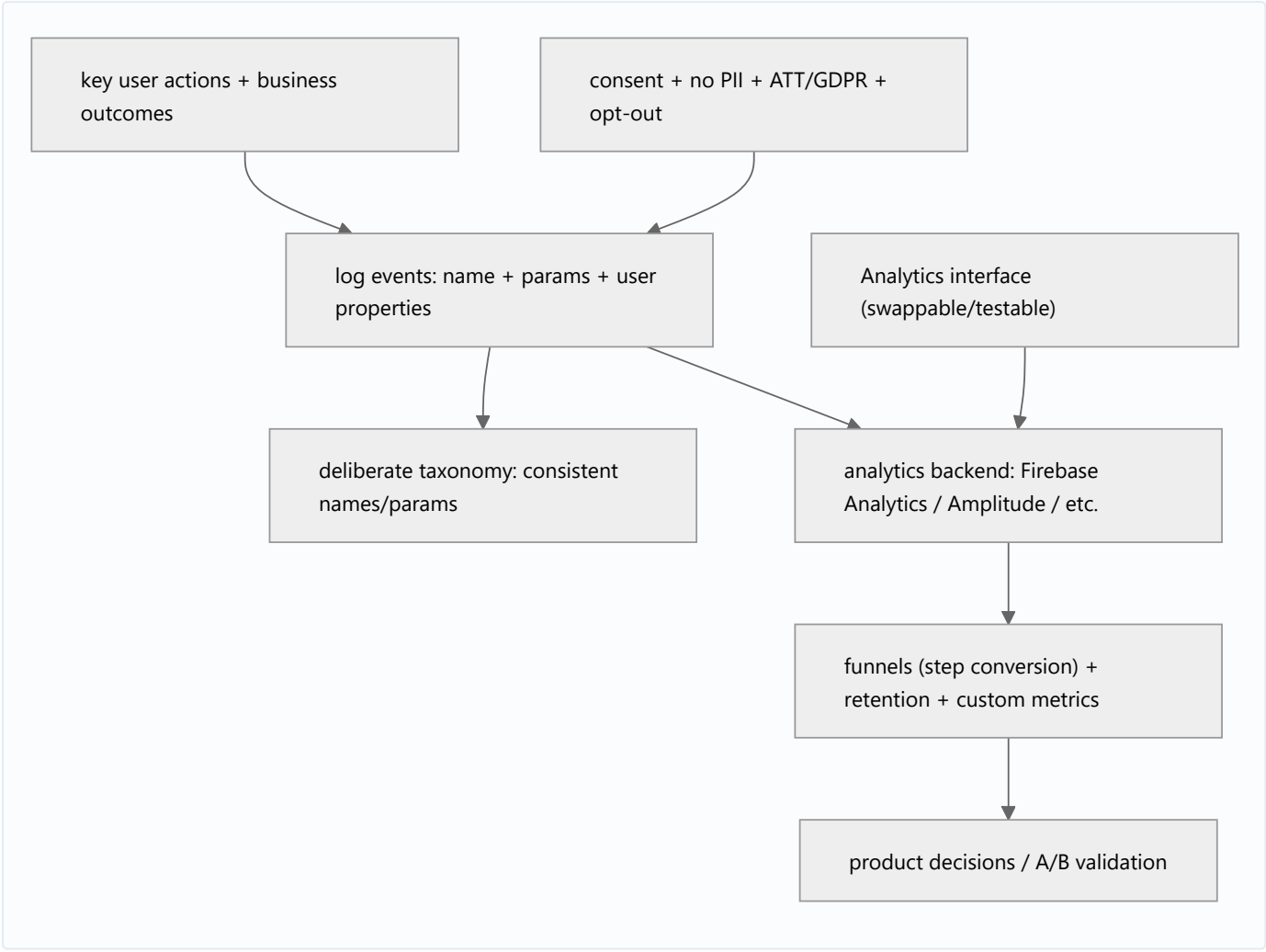
Why this concept exists

Building features without measuring usage is guessing — you can't tell what's used, where users drop off, or whether a change helped. Analytics turns product decisions from opinion into **data**: funnels reveal drop-offs, retention shows stickiness, A/B metrics validate changes. Done with a clear taxonomy + privacy discipline, it's a reliable decision tool; done ad hoc, it's noisy, unanalyzable, and a compliance risk.

Real-world analogy

Analytics is a **store's foot-traffic + checkout study**: you note where shoppers **enter, browse, add to cart, and pay** (events/funnel), how many **come back** (retention), and which displays convert (custom metrics) — to **rearrange the store** based on evidence, not hunches. But you **don't record identities** (privacy), and you use a **consistent tally sheet** (taxonomy) so the numbers are comparable, and you **ask consent** before tracking.

Internal Working



- **Events + parameters + user properties**:
  - **Events**: discrete user/business actions (`screen_view` , `search` , `add_to_cart` , `checkout_start` , `purchase` , `signup`). Each has **parameters** (`item_id` , `value` , `method`) for slicing.
  - **User properties**: durable attributes for segmentation (`plan: pro` , `locale` , `signup_cohort`) — **not PII**.
  - Some events are **auto-collected** (screen views, session start) by SDKs; you add **custom** events for what matters.
- **Event taxonomy (the discipline)**: define a **consistent, documented** naming/parameter scheme **before** instrumenting (snake_case names, standard params, an event dictionary). Ad hoc/inconsistent events (`AddCart` vs `add_to_cart` vs `cart_add`) are **unanalyzable**. Keep it **minimal + meaningful** — instrument **key actions + outcomes**, not every tap (noise + cost + privacy risk).
- **Analysis**:
  - **Funnels**: ordered steps (view → add_to_cart → checkout → purchase) showing **conversion + drop-off** per step — the core product insight.
  - **Retention**: do users **come back** (D1/D7/D30)? Cohorts by signup date/feature.

    - **Custom metrics/dashboards**: business KPIs (activation rate, feature adoption, revenue events).
    - **A/B/experiments**: measure whether a change moves the metric (via Firebase Remote Config/experiments or a dedicated tool).
- **Privacy/consent (non-negotiable)** (Module 37/Module 39):
    - **No PII** in events/properties (no emails/names/precise location); use **opaque ids**; hash where correlation is needed.
    - **Consent**: honor **ATT (iOS tracking prompt)**, **GDPR/CCPA** consent (opt-in/opt-out), and platform data-safety declarations (must match — Module 51). Provide an **opt-out**.
    - **Data minimization**: collect only what informs decisions.
- **Abstraction (swappable/testable)**: put analytics behind an `Analytics` **interface** (`logEvent(name, params)`, `setUserProperty`) with the vendor SDK as one impl — so you can **swap tools**, **fake in tests**, and **centralize redaction/consent gating**. App code calls the interface, not the SDK directly (Module 39 facade pattern).
- **Analytics ≠ health monitoring**: analytics = **product usage/learning** (what/why users do); crash/perf = **health**. Both are pillars but answer different questions — don't conflate (or over-collect).
- **Tools**: **Firebase Analytics** (free, Flutter-integrated, funnels/retention, ties to Crashlytics/Remote Config), **Amplitude/Mixpanel** (deeper product analytics), backend event pipelines. Wire behind the interface.

Memory Representation

Not app state — an **event stream + user-property store** in the analytics backend, aggregated into funnels/retention/metrics. On-device: a small batched queue of events (async upload); consent flags gate emission. The event **taxonomy** is a documented dictionary.

Compiler Behavior

Analytics is normal code + SDK; the `Analytics` interface lets the compiler enforce dependency on the abstraction (swappable/fake-able).

Runtime Behavior

Events are queued + uploaded **async/batched** (low overhead); consent gating drops events when not permitted; the backend aggregates into funnels/retention (often with reporting latency — not real-time).

Flutter Engine Behavior

Screen-view auto-collection may hook navigation/route observers; otherwise analytics is app-level.

Dart VM Behavior

Not applicable beyond batching being off the critical path.

Examples

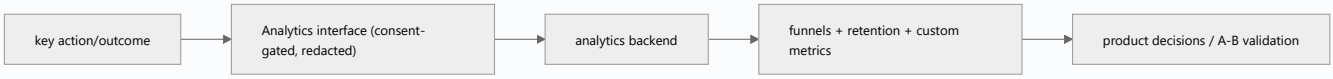
```
// Analytics behind an interface (swappable, testable, consent-gated, redacted)
abstract class Analytics {
  Future<void> logEvent(String name, [Map<String, Object?> params = const {}]);
  Future<void> setUserProperty(String key, String value);
}

class FirebaseAnalyticsImpl implements Analytics {
  final FirebaseAnalytics _fa; final Consent _consent;
  FirebaseAnalyticsImpl(this._fa, this._consent);
  @override
  Future<void> logEvent(String name, [Map<String, Object?> params = const {}]) async {
    if (!_consent.analyticsAllowed) return;               // consent gate
    await _fa.logEvent(name: name, parameters: _redact(params).cast()); // no PII
  }
  @override
  Future<void> setUserProperty(String k, String v) =>
      _consent.analyticsAllowed ? _fa.setUserProperty(name: k, value: v) : Future.value();
}

// Deliberate taxonomy — consistent names/params for a funnel
analytics.logEvent('view_item', {'item_id': id, 'category': cat});
analytics.logEvent('add_to_cart', {'item_id': id, 'value': priceCents});
analytics.logEvent('checkout_start', {'cart_size': n});
analytics.logEvent('purchase', {'value': totalCents, 'currency': 'USD'});  // business outcome
// -> funnel: view_item -> add_to_cart -> checkout_start -> purchase (drop-off per step)
```

```
Event dictionary (documented taxonomy — snake_case, standard params):
  view_item {item_id, category}
  add_to_cart {item_id, value}
  checkout_start {cart_size}
  purchase {value, currency}          # business outcome
  User properties: plan (free|pro), locale, signup_cohort   (NO PII)
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Ad hoc/inconsistent event names	Unanalyzable data	Deliberate documented taxonomy (snake_case, standard params)
Logging everything	Noise + cost + privacy risk	Instrument key actions + outcomes only
PII in events/properties	Privacy/compliance breach	Opaque ids; no PII; redact
Ignoring consent/ATT/GDPR	Illegal tracking / rejection	Consent-gate; honor ATT/GDPR; opt-out
Calling the SDK directly everywhere	Not swappable/testable	`Analytics` interface abstraction
Conflating analytics with health monitoring	Wrong tool/questions	Analytics = usage; crashes/perf = health
Never acting on funnels	Data without decisions	Close the loop: analyze → decide/experiment
Mismatched data-safety declarations	Store rejection/removal	Declarations match collected events

Best Practices

- Define a **deliberate, documented event taxonomy** (consistent names/params, an event dictionary) and instrument **key actions + business outcomes** — minimal + meaningful, not everything.
- Analyze via **funnels** (conversion/drop-off), **retention**, and **custom metrics**; **close the loop** (decisions/experiments), not just dashboards.

- Enforce **privacy/consent**: **no PII** (opaque ids), honor **ATT/GDPR/CCPA** (opt-in/opt-out), **data-minimize**, and match **store data-safety declarations**.
- Keep analytics **behind an interface** (swappable, testable, centralized consent/redaction); treat analytics (usage) as **distinct from health monitoring**.

Performance

Analytics is low-overhead: events are batched + async, consent-gated, and sampled if high-volume — must not tax CPU/battery/data. Backend reporting has latency (not real-time). The cost of getting it wrong is privacy/compliance (leaked PII, no consent) more than performance.

Advantages / Disadvantages

- **+** Data-driven product decisions (funnels/retention/experiments), feature-adoption insight, swappable/testable (interface), privacy-respecting.
- **−** Taxonomy discipline + upkeep, privacy/consent complexity (ATT/GDPR), backend cost, easy to over-collect (noise/risk), reporting latency.

Interview Questions

1. 🟢 **What does analytics measure, vs crash/perf monitoring?** — Analytics = product usage/behavior (what/why users do: events/funnels/retention) for decisions; crash/perf = app health (stability/performance).
2. 🟢 **Why is an event taxonomy important?** — Consistent, documented names/params make data analyzable; ad hoc events (`AddCart` vs `add_to_cart`) can't be aggregated meaningfully.
3. 🟡 **What is a funnel, and why is it the core insight?** — An ordered sequence of steps (view→cart→checkout→purchase) showing conversion + drop-off per step, revealing where users abandon.
4. 🟡 **How do you handle privacy/consent in analytics?** — No PII (opaque ids), honor ATT/GDPR/CCPA (opt-in/opt-out), data-minimize, match store data-safety declarations; consent-gate emission.
5. 🟡 **Why put analytics behind an interface?** — Swappability (change vendors), testability (fake), and a single place for consent gating + redaction — app code depends on the abstraction.
6. 🔴 **What should you instrument, and what not?** — Key user actions + business outcomes (meaningful, minimal); not every tap (noise + cost + privacy risk).
7. 🔴 **How do you avoid analytics being wasted?** — Close the loop: analyze funnels/retention → make product decisions / run experiments; dashboards nobody acts on are wasted.

Senior Engineer Tips

- Design the event taxonomy + dictionary before instrumenting and enforce it in review; inconsistent event names are the #1 reason analytics data is unusable later.
- Route all analytics through an interface that gates on consent and redacts, so PII can't leak and you can swap Firebase↔Amplitude without touching features.
- Instrument outcomes + funnel steps that map to decisions, and actually review them; a wall of vanity events you never analyze is cost + privacy risk with no payoff.

Architect Perspective

Analytics is the product-learning pillar of observability: a deliberate event taxonomy feeding funnels/retention/custom metrics that drive decisions, kept privacy-safe and behind a swappable interface. Distinct from health monitoring (crashes/perf), it answers "what/why users do" and closes the product feedback loop (measure → decide → experiment). Architecturally it's the same discipline as logging — an abstraction, consent/redaction centralized, minimal + meaningful — applied to behavior rather than diagnostics (01_monitoring_fundamentals.md, Module 39, Module 37).

Summary

- Analytics logs deliberate events (taxonomy) + user properties to analyze funnels/retention/custom metrics for product decisions — distinct from health monitoring.
- Instrument key actions + outcomes (minimal/meaningful); enforce privacy/consent (no PII, ATT/GDPR, opt-out, match data-safety); keep behind a swappable interface.
- Close the loop (analyze → decide/experiment); low-overhead (async/batched/consent-gated).

Revision Notes

- Events (name + params) + user properties (opaque, no PII); auto + custom; deliberate documented taxonomy (snake_case, standard params, event dictionary); instrument key actions + business outcomes (minimal).
- Analyze: funnels (conversion/drop-off), retention (D1/D7/D30 cohorts), custom metrics/dashboards, A/B experiments (Remote Config). Analytics = usage (decisions), not health.
- Privacy: no PII, consent (ATT/GDPR/CCPA opt-in/out), data-minimize, match store data-safety; behind `Analytics` interface (swappable/testable/consent-gated/redacted); async/batched. Tools: Firebase Analytics/Amplitude/Mixpanel.

Practice Questions

1. Why is a consistent event taxonomy essential?
2. How do privacy/consent constrain analytics, and how do you comply?
3. Why wrap analytics behind an interface?

Coding Questions

1. Define an `Analytics` interface + a consent-gated, redacted vendor impl.
2. Instrument a purchase funnel with a consistent taxonomy.
3. Add a user property + a custom business-outcome event (PII-safe).

Mini Project

Product analytics (Flutter/monitoring): For an e-commerce-ish flow, define an event taxonomy + dictionary (view_item → add_to_cart → checkout_start → purchase + user properties), instrument it behind an `Analytics` interface (consent-gated + redacted vendor impl, no PII), and describe the funnel/retention analysis + a resulting product decision/experiment. Ensure ATT/GDPR consent + data-safety alignment. Acceptance: deliberate taxonomy (consistent/documented); key actions + outcomes instrumented (minimal); behind a swappable/testable interface; consent-gated + PII-safe + data-safety-aligned; funnel + a data-driven decision; analytics distinguished from health monitoring.

Performance Monitoring & Alerting

> Lab performance (DevTools on your device) ≠ **field performance** (thousands of real devices/networks) — so monitor it in production: **startup time**, **frame rendering** (slow/frozen frames, jank %), **network** (latency, error rate), and **custom traces** (a checkout's duration), via **Firebase Performance/Sentry Performance** or custom metrics. Turn raw numbers into **SLIs** (Service Level Indicators — the measured signals, tracked at **p50/p90/p95/p99**, not averages) with **SLOs** (targets, e.g., "p95 cold start < 2s, 99.5% crash-free"), surfaced on **dashboards** and guarded by **alerts** on **SLO breaches/regressions** — so a bad release or degradation is caught (and staged rollout halted) before wide impact.

Introduction

This file covers field performance monitoring + alerting: what to measure (startup/frames/network/custom traces), percentiles vs averages, SLIs/SLOs, dashboards, and alerting (including rollout gating). It extends the performance discipline (Module 21/Module 47) into production.

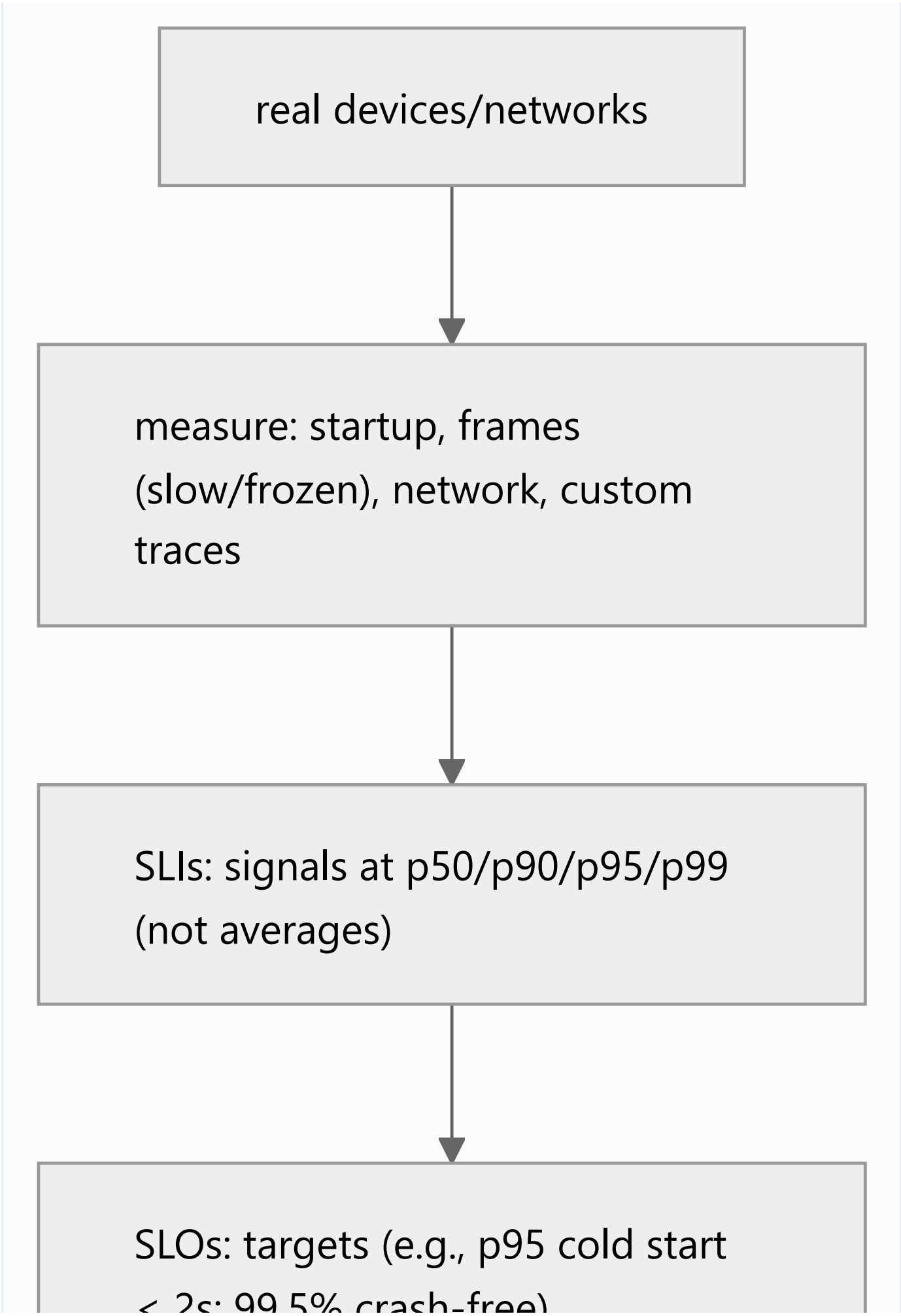
Why this concept exists

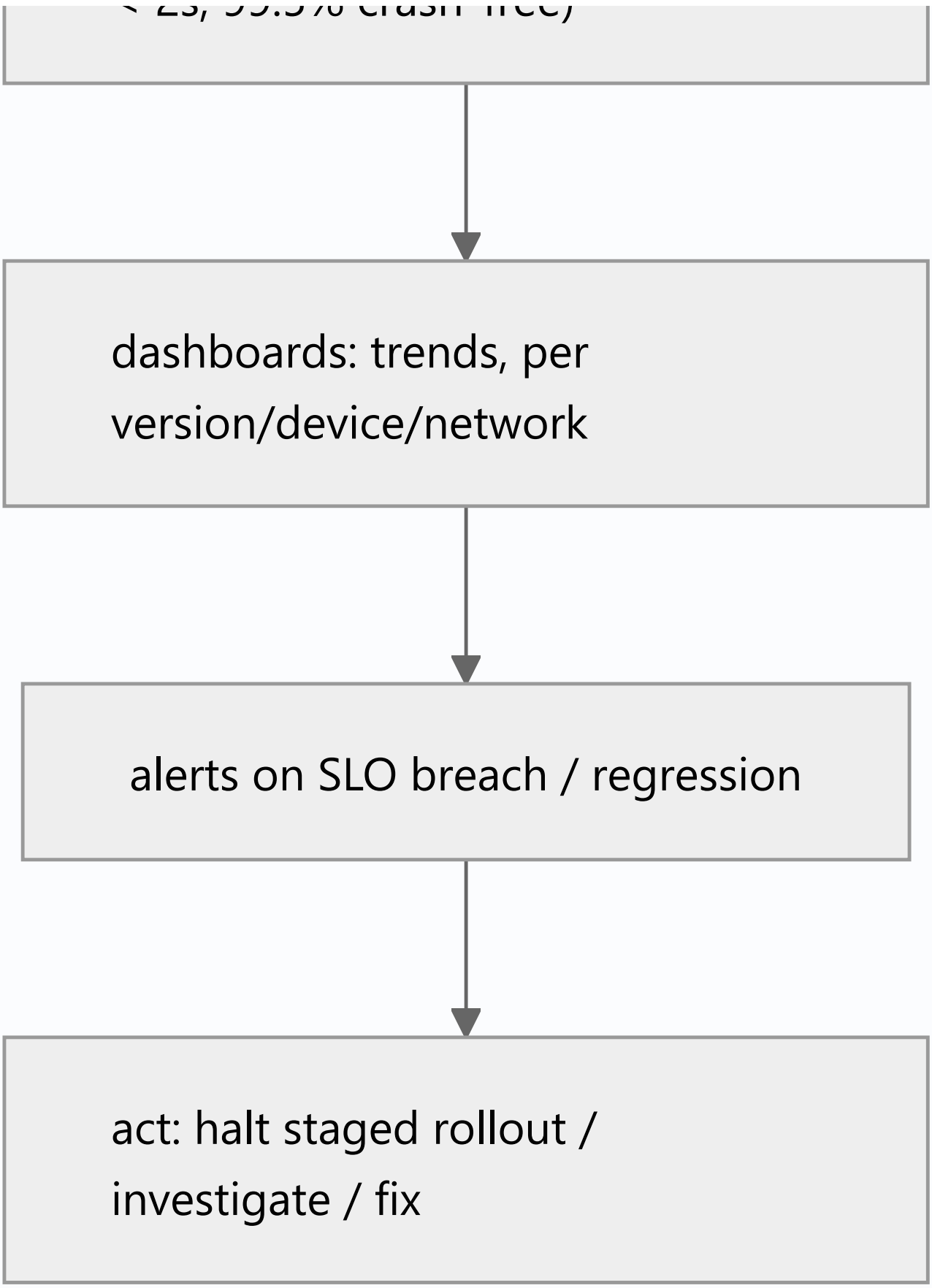
An app can be smooth on your flagship dev phone and janky/slow on a mid-range device on a 3G network — you only learn this from **real-world data**. And performance **regresses silently** as features land. Field monitoring + SLOs + alerts make performance an **enforced, observable** property (like crash-free rate for stability), catching regressions in production and gating releases.

Real-world analogy

It's **fleet telemetry vs a single test drive**: your one test car (dev device) says "smooth," but the **whole fleet in real conditions** (traffic, weather, terrain) tells the truth — so you monitor **fuel efficiency, engine temp, braking distance** across all vehicles at the **95th percentile** (worst realistic, not the average). You set **targets (SLOs)**, watch a **dashboard**, and get **alerts** when a model's metrics degrade — pulling that model from wide release before customers are affected.

Internal Working


real devices/networks
measure: startup, frames (slow/frozen), network, custom traces
SLIs: signals at p50/p90/p95/p99 (not averages)
SLOs: targets (e.g., p95 cold start



- **What to measure (mobile field perf)**:

  - **Startup time**: **cold/warm/hot** start, time-to-first-frame/interactive — a top UX + install-retention metric.
  - **Frame rendering**: **slow frames** (>16ms@60 / >8ms@120) and **frozen frames** (>700ms) → **jank %**; the smoothness signal (Module 21).
  - **Network**: request **latency**, **error/timeout rate**, payload sizes — per endpoint.
  - **Custom traces**: wrap key operations (checkout, search, image load) to measure **duration + success** in the field.
  - **Resource**: memory/ANRs/battery where available.
- **Percentiles, not averages (critical)**: report **p50/p90/p95/p99** — the **average hides the tail** (a 2s mean can hide many 8s experiences). SLIs/SLOs target **high percentiles** (p95/p99) because that's where users suffer. Slice by **version/device tier/network/region** to find who's affected.
- **SLIs / SLOs / (SLAs)**:
  - **SLI** = the **measured signal** (e.g., p95 cold-start time, crash-free rate, p95 checkout duration).
  - **SLO** = the **target/objective** for an SLI (e.g., "p95 cold start < 2s", "99.5% crash-free", "checkout p95 < 3s"). Your internal quality bar.
  - **SLA** = a **contractual** guarantee (mostly backend/enterprise); mobile teams mostly use SLIs/SLOs internally.
  - SLOs make performance **measurable + enforceable**, not vibes.
- **Dashboards**: visualize SLIs over time, **per version** (release health), device tier, network — to spot **trends + regressions** (a new version's p95 startup climbing) alongside crash-free rate (02_crash_reporting.md).
- **Alerting (the action trigger)**:
  - Alert on **SLO breaches** (crash-free < target, p95 startup > target), **regressions** (metric worse than the previous release), and **spikes** (error-rate/jank surge) — especially **during staged rollout**.
  - **Tune to avoid noise**: alert on **sustained, meaningful** breaches (not single blips); route to the right channel; make each alert **actionable** (maps to a decision: halt rollout / investigate / hotfix).
- **Rollout gating (closes with deployment)**: during **staged/phased rollout**, monitored SLIs (crash-free + perf) decide **promote vs halt** — a regression at 5% stops it before 100% (Module 51).
- **Tools**: **Firebase Performance Monitoring** (auto startup/frame/network traces + custom traces, Flutter-integrated), **Sentry Performance** (transactions/spans + tracing), custom metrics to a backend. Wire behind an interface where practical; keep **low-overhead** (sampling).

- **Overhead + privacy**: performance SDKs sample + batch (low overhead); attach only **non-PII** context (version/device/network) (Module 37).

Memory Representation

Not app state — **time-series SLIs** (percentile distributions per version/device/network) + custom-trace spans in the monitoring backend; SLO targets + alert rules are config. On-device: sampled trace/metric collection, batched async.

Compiler Behavior

Not applicable (SDK-based). Custom traces are code instrumentation; performance data ties to app version/build for release comparison.

Runtime Behavior

The SDK samples startup/frame/network + custom traces during real use, batches async (low overhead); the backend aggregates percentiles + fires alerts on breaches/regressions. Rollout decisions read these live.

Flutter Engine Behavior

Frame/startup traces hook engine timings (build/layout/paint/raster + time-to-first-frame); slow/frozen-frame classification comes from the rendering pipeline (Module 09/Module 21).

Dart VM Behavior

Custom traces measure Dart operation durations; isolate/GC pauses can surface as jank in frame metrics. Sampling keeps overhead low.

Examples

```
// Custom trace (Firebase Performance) — measure a key operation in the field
final trace = FirebasePerformance.instance.newTrace('checkout');
await trace.start();
try {
  await performCheckout();
  trace.putAttribute('result', 'success');
} catch (_) {
  trace.putAttribute('result', 'failure');
} finally {
  await trace.stop();   // duration recorded across real devices/networks
}
// Firebase auto-collects: app start time, slow/frozen frames, HTTP/S network traces.
```

```
SLIs / SLOs (percentiles, not averages):
  SLI: p95 cold start time      SLO: < 2.0s
  SLI: crash-free users         SLO: >= 99.5%
  SLI: jank % (slow/frozen)     SLO: < 1% of frames
  SLI: p95 checkout duration    SLO: < 3.0s
  SLI: network error rate       SLO: < 1%
  -> report p50/p90/p95/p99, sliced by version/device tier/network; alert on breach/regression

Alerting (actionable, tuned):
  crash-free rate < 99.5% (sustained)      -> HALT staged rollout, investigate
  p95 startup > 2s on new version          -> regression alert -> hold promotion
  network error rate spike on endpoint X   -> investigate backend/contract
```

Diagrams

Common Mistakes

Mistake	Why it's wrong	Fix
Only lab (dev-device) performance	Misses field reality (mid-range/3G)	Monitor in production across devices/networks
Averages instead of percentiles	Hides the painful tail	Track p95/p99 (+ p50/p90)
No SLIs/SLOs	Performance is "vibes"	Define SLIs + SLO targets
No per-version dashboards	Miss release regressions	Slice by version (release health)
Alerting on single blips	Alert fatigue → ignored	Alert on sustained/meaningful breaches, actionable
Not gating rollout on perf/crash	Bad release hits everyone	Halt staged rollout on SLO breach
Heavy/unsampled tracing	Overhead	Sample + batch (low overhead)
PII in trace attributes	Privacy breach	Non-PII context only

Best Practices

- **Monitor performance in the field** (startup, slow/frozen frames + jank %, network latency/errors, custom traces) across **devices/networks/versions** — not just on your dev device.
- Use **percentiles (p95/p99)**, not averages; define **SLIs + SLOs** (targets) so performance is measurable/enforceable; surface on **per-version dashboards** alongside crash-free rate.
- **Alert on SLO breaches/regressions/spikes** (sustained, actionable, tuned to avoid noise) and **gate staged rollout** on them (halt on regression before wide impact).
- Keep monitoring **low-overhead** (sampling/batching) and **PII-safe**; wrap key operations in **custom traces**; act on alerts (close the loop).

Performance

Ironically, monitoring must itself be **low-overhead** (sampled/batched) — but its payoff is protecting field performance: catching startup/frame/network regressions the lab misses, enforcing SLOs, and gating releases. Percentile tracking focuses effort where users actually hurt (the tail) (Module 21/Module 47).

Advantages / Disadvantages

- **+** Real-world performance visibility, percentile/tail insight, per-version regression detection, SLO enforcement, rollout gating, actionable alerts.
- **−** SDK/backend setup + cost, alert tuning (noise vs signal), SLO definition/maintenance, sampling trade-offs, low-overhead + privacy discipline.

Interview Questions

1. 🟢 **Why monitor performance in production, not just DevTools?** — Lab (your device) ≠ field (thousands of real devices/networks); startup/jank/network vary hugely and regress silently — only field data reveals the truth.
2. 🟢 **Why percentiles instead of averages?** — Averages hide the tail; p95/p99 capture the painful worst-case experiences where users actually suffer.
3. 🟡 **What are SLIs and SLOs?** — SLI = the measured signal (p95 cold start, crash-free rate); SLO = the target for it (e.g., p95 < 2s, ≥99.5% crash-free) — making performance measurable/enforceable.
4. 🟡 **What field metrics do you track for a Flutter app?** — Startup time, slow/frozen frames (jank %), network latency/error rate, custom operation traces, memory/ANRs — sliced by version/device/network.
5. 🟡 **How do alerts avoid noise while staying useful?** — Alert on sustained/meaningful SLO breaches/regressions (not single blips), route them, and make each actionable (maps to a decision like halt rollout).
6. 🔴 **How does performance monitoring gate deployment?** — During staged rollout, SLI regressions (crash-free/perf) trigger halting/rolling back before 100% exposure (Module 51).
7. 🔴 **How do you keep monitoring itself cheap + private?** — Sample + batch traces (low overhead) and attach only non-PII context (version/device/network).

Senior Engineer Tips

- Define a few meaningful SLIs/SLOs (crash-free rate, p95 cold start, jank %, key custom-trace durations) and put them on a per-version dashboard; that's your release-health cockpit.
- Track percentiles and slice by device tier/network — the average makes a broken mid-range/3G experience invisible, which is exactly the segment that churns.
- Make every alert actionable and tuned; alert on sustained SLO breaches during rollout so you halt before 100%, and delete noisy alerts that trained everyone to ignore them.

Architect Perspective

Performance monitoring + alerting is the performance sensor of observability: it moves performance from a lab hope to an enforced, field-measured property via SLIs/SLOs at high percentiles, per-version dashboards, and actionable alerts that gate staged rollouts. Together with crash reporting (stability) it forms the release-health signals that make rollouts safe and regressions catchable — the production enforcement of the performance budgets defined earlier (Module 21, Module 47, Module 51, 02_crash_reporting.md).

Summary

- Monitor field performance (startup, slow/frozen frames + jank %, network, custom traces) across devices/networks/versions — lab ≠ field.
- Use percentiles (p95/p99); define SLIs + SLOs (targets); dashboard per version; alert on breaches/regressions (sustained, actionable) and gate staged rollout.
- Keep monitoring low-overhead (sampled/batched) + PII-safe; act on alerts (close the loop).

Revision Notes

- Measure (field, not lab): startup (cold/warm/hot, TTFF), frames (slow >16ms/frozen >700ms → jank %), network (latency/error rate), custom traces (checkout/search), memory/ANRs. Firebase Performance/Sentry Performance.
- Percentiles (p50/p90/**p95/p99**) not averages (tail hidden); slice by version/device tier/network. SLI = measured signal; SLO = target (p95 cold start <2s, ≥99.5% crash-free, jank <1%); SLA = contractual (backend).
- Dashboards per version (release health, w/ crash-free); alerts on SLO breach/regression/spike (sustained + actionable + tuned) → gate staged rollout (halt on regression). Low-overhead (sample/batch) + PII-safe context.

Practice Questions

1. Why report p95/p99 instead of averages?
2. Define SLI vs SLO with mobile examples.
3. How do performance/crash SLIs gate a staged rollout?

Coding Questions

1. Add a custom trace around a key operation with a success attribute.
2. Define SLIs/SLOs for startup, frames, and a custom trace.
3. Specify alert rules (breach/regression) that halt a staged rollout.

Mini Project

Field performance monitoring (Flutter/monitoring): Wire performance monitoring (Firebase Performance/Sentry): auto startup/frame/network traces + a custom trace around a key flow (e.g., checkout), define SLIs/SLOs (p95 cold start, jank %, crash-free rate, custom-trace p95) tracked at percentiles per version/device/network, build a release-health dashboard, and set actionable alerts (SLO breach/regression) that gate staged rollout — low-overhead + PII-safe. Acceptance:

field metrics (startup/frames/network/custom) at percentiles, sliced by version/device/network; SLIs + SLO targets; per-version dashboard (+ crash-free); tuned actionable alerts → rollout halt; low-overhead (sampled/batched) + non-PII context.

Monitoring Integration (Capstone: Full Observability + Feedback Loop)

> Assemble one **observability stack** that closes the loop: **crash reporting** (Crashlytics/Sentry — crash-free rate, symbolication, breadcrumbs), **analytics** (event taxonomy + funnels, privacy-safe), and **performance monitoring** (startup/frame/network SLIs, SLOs), all **correlated** (shared ids + version/device/session context), fed by your **logging facade**, surfaced on **dashboards**, and guarded by **actionable alerts** — wired into the **delivery feedback loop**: staged-rollout gate (crash-free + SLO) → triage → fix/hotfix → iterate. This completes the build→ship→**observe**→**improve** cycle and makes every earlier module's work verifiable in production.

Introduction

This module capstone composes crash reporting, analytics, and performance monitoring into a single correlated, privacy-safe, low-overhead observability stack that drives action — the "how it all fits + closes the loop" deliverable. It ties monitoring to deployment (staged rollout) and to the whole handbook (verifying architecture/perf/quality in the field).

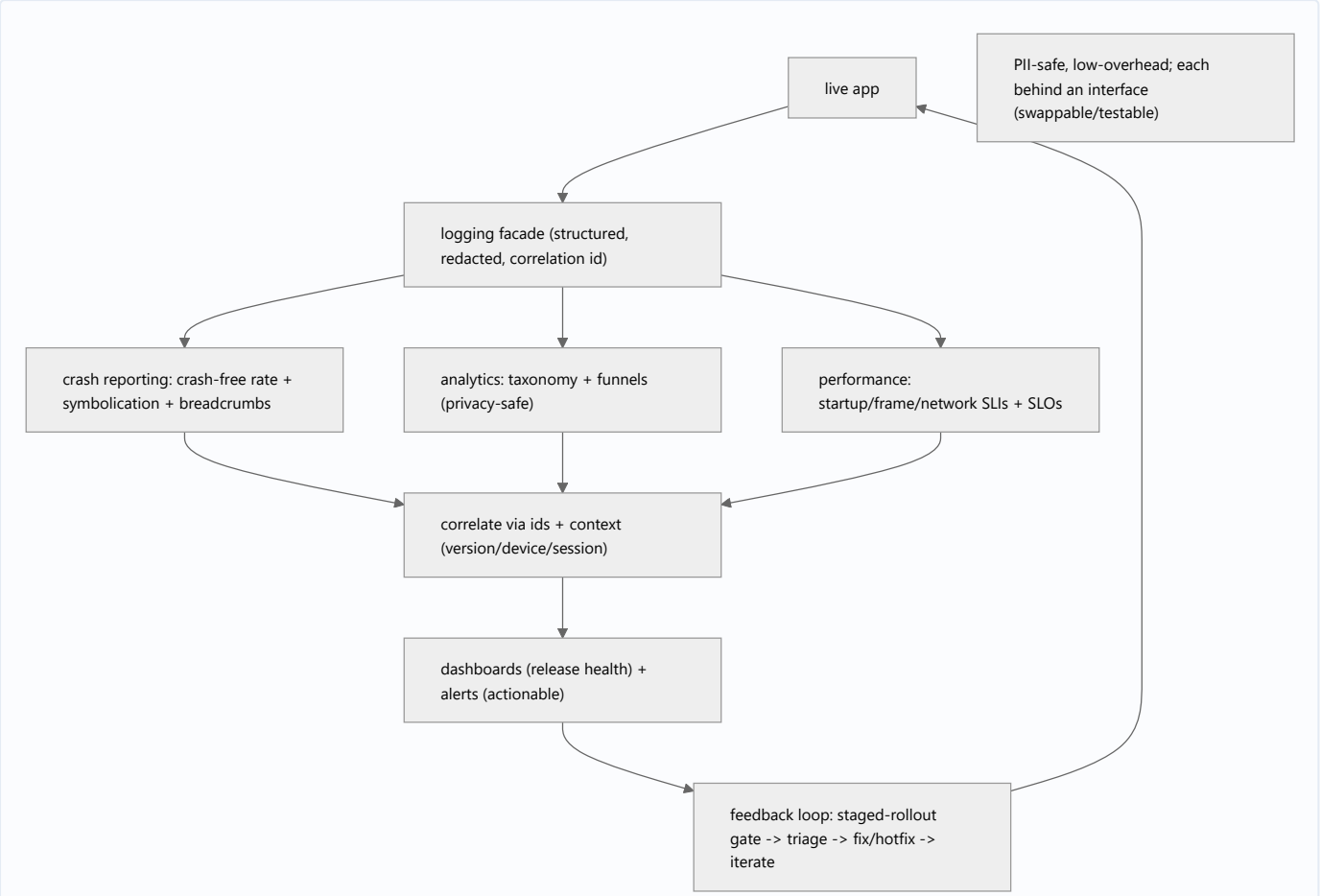
Why this concept exists

The pillars are only valuable **assembled + correlated + acted upon**: crashes, analytics, and performance answer different questions but must stitch together (one correlation id linking a crash to its funnel step + trace), surface on shared dashboards, alert meaningfully, and **feed decisions** (halt rollout, hotfix, iterate). This capstone shows that integrated stack and the feedback loop that makes monitoring worthwhile.

Real-world analogy

It's **mission control for a live product**: separate consoles (stability, usage, performance) feed **one correlated picture** with shared telemetry ids; a **health board** shows release status; **alarms** fire on real anomalies; and controllers **act** — hold the launch (halt rollout), send a fix (hotfix), or plan the next mission (iterate). Disconnected consoles nobody watches or acts on aren't mission control — the **integrated, acted-upon loop** is.

Internal Working



- **Three pillars, correlated** (01_monitoring_fundamentals.md):
  - **Stability** — crash reporting (crash-free rate, symbolicated stacks, breadcrumbs — 02_crash_reporting.md).
  - **Usage** — analytics (event taxonomy, funnels, retention, privacy-safe — 03_analytics_and_metrics.md).
  - **Performance** — field SLIs/SLOs (startup/frames/network/custom traces — 04_performance_monitoring_and_alerting.md).
  - **Correlated** by a shared **correlation/trace id** + **context** (version/device/session/opaque user) so a crash links to its funnel step + trace + breadcrumbs.
- **Fed by the logging facade** (Module 39): one place emits structured, redacted, correlated telemetry to all three (breadcrumbs → crash reporter; events → analytics; spans → performance) — keeping instrumentation **DRY, PII-safe, consent-gated**.
- **Behind interfaces (swappable/testable)**: `CrashReporter`, `Analytics`, `PerformanceMonitor` abstractions so vendors are swappable, tests use fakes, and consent/redaction is centralized — the same discipline as logging.
- **Dashboards + alerts**:
  - A **release-health dashboard** (crash-free rate + key SLIs per version) is the launch cockpit.

    - **Actionable, tuned alerts** on **crash-free-rate drops, SLO breaches/regressions, funnel-conversion drops, error spikes** — routed + mapped to a decision.
- **The feedback loop (the whole point)** (Module 51):
    - **Staged-rollout gate**: monitored signals (crash-free + SLOs) decide **promote vs halt** at 5%/20%/… — a regression stops it before 100%.
    - **Triage → fix/hotfix**: alert → open the correlated issue (crash stack + breadcrumbs + trace + funnel context) → fix → **forward-fix/hotfix** (mobile rollback is hard — flags/kill switch + expedited release).
    - **Iterate**: funnels/retention + perf/stability trends → next release; **re-verify** metrics recover.
- **Verifies the whole handbook in the field**: monitoring confirms your **architecture/perf/quality** actually hold on real devices — crash-free rate validates error handling/testing, SLIs validate performance budgets, funnels validate product decisions. It's the **outer verification loop** of everything built.
- **Constraints** (Module 37): **PII-safe** (redact, opaque ids, consent — ATT/GDPR, matching store data-safety), **low-overhead** (async/batched/sampled) across all pillars.
- **Right-sizing**: a small app needs crash reporting + a couple of SLIs + basic analytics; a large app adds full dashboards, tuned alerting, SLO governance, on-call, and experiment analytics — scale to the product (Module 47).

Memory Representation

Not app state — a **correlated telemetry system**: crash issues, analytics event streams/funnels, performance SLI time-series, all keyed by correlation ids + context, surfaced on dashboards with alert rules. On-device: the logging facade batches/redacts/consent-gates telemetry to the three sinks (async).

Compiler Behavior

Instrumentation compiles against the pillar interfaces (mockable); release obfuscation → symbolication via archived mapping (Module 51). Global handlers are entry points feeding crash reporting.

Runtime Behavior

The app emits correlated telemetry (async/batched/sampled); backends aggregate into crash-free rate, funnels, and SLIs; alerts fire on anomalies; rollout decisions + triage + iteration act on them. Correlation stitches a single incident across pillars.

Flutter Engine Behavior

Performance hooks engine frame/startup timings; crash SDKs capture framework/native errors; analytics may observe navigation — all funneled through the facade.

Dart VM Behavior

Uncaught Dart/isolate errors → crash reporter (via global handlers); custom traces measure Dart operations; batching keeps all pillars off the critical path.

Examples

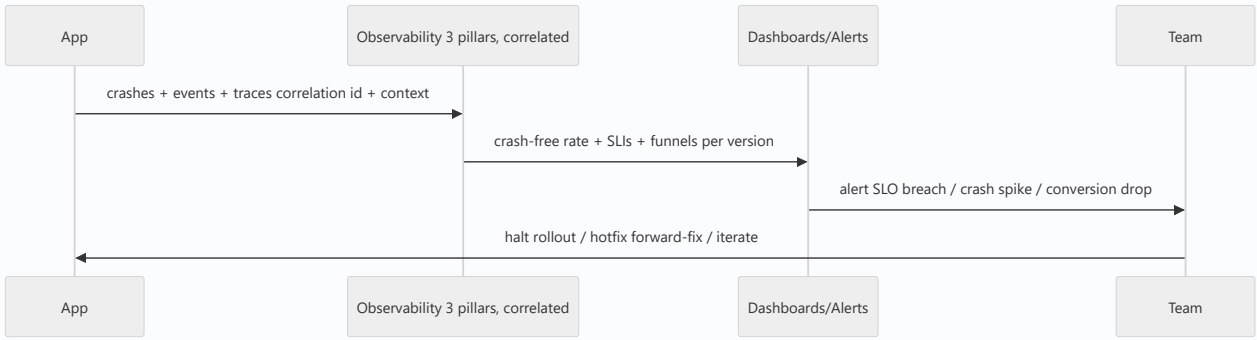
```
// One facade feeds all three pillars, correlated + redacted + consent-gated
class Observability {
  final CrashReporter crashes; final Analytics analytics; final PerformanceMonitor perf;
  Observability(this.crashes, this.analytics, this.perf);

  void bootstrap(String corrId, String version) {
    crashes.setContext({'version': version, 'correlationId': corrId}); // ties pillars together
    FlutterError.onError = crashes.recordFlutterError;                 // stability
    PlatformDispatcher.instance.onError = (e, s) { crashes.record(e, s, fatal: true); return true; };
  }

  Future<void> checkout(Cart cart) async {
    final trace = perf.startTrace('checkout');                         // performance
    analytics.logEvent('checkout_start', {'cart_size': cart.items.length}); // usage (funnel)
    try {
      await doCheckout(cart);
      analytics.logEvent('purchase', {'value': cart.totalCents});
    } catch (e, s) {
      crashes.record(e, s, fatal: false);                              // non-fatal (stability)
      analytics.logEvent('checkout_failed');
      rethrow;
    } finally {
      await trace.stop();                                              // duration SLI
    }
  }
}
// All three share the correlation id + version context -> one incident view.
```

```
Feedback loop (staged rollout -> action):
  release at 5% -> watch crash-free rate + SLIs + funnel conversion
    if crash-free drops / SLO breach / conversion tanks -> HALT + triage (correlated:
crash+breadcrumbs+trace+funnel)
    -> forward-fix/hotfix (flags/kill switch + expedited release) -> verify recovery
    else -> promote 20% -> ... -> 100% -> iterate next release on funnel/perf/stability data
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Pillars uncorrelated	Can't stitch a full incident	Shared correlation id + context
Collecting but not acting	Monitoring without the loop is waste	Gate rollout / triage / fix / iterate
Direct SDK calls everywhere	Not swappable/testable/consistent	Interfaces + logging facade
PII across telemetry	Privacy/compliance breach	Redact, opaque ids, consent (all pillars)
No release-health dashboard/alerts	Regressions unnoticed	Per-version dashboard + actionable alerts
Not gating staged rollout	Bad release hits everyone	Halt on crash-free/SLO regression
High-overhead telemetry	Harms the app	Async/batched/sampled across pillars
Over-monitoring a tiny app	Overhead > value	Right-size (crashes + few SLIs + basic analytics)

Best Practices

- Integrate **all three pillars** (stability/usage/performance) behind **interfaces**, fed by the **logging facade**, **correlated** via shared ids + context (version/device/session/opaque user).
- Surface a **release-health dashboard** (crash-free rate + key SLIs + funnels per version) and set **actionable, tuned alerts** (crash-free drop / SLO breach / conversion drop / error spike).

- **Close the feedback loop**: gate **staged rollout** on signals → triage the correlated incident → **forward-fix/hotfix** (flags/kill switch) → **iterate**; re-verify recovery.
- Keep telemetry **PII-safe + consent-gated + low-overhead** across pillars; **right-size** to the product; use monitoring to **verify** your architecture/perf/quality in the field.

Performance

The stack is designed low-overhead (async/batched/sampled across pillars); its payoff is protecting field **stability + performance** and enabling safe rollouts. Correlation makes diagnosis fast (fewer wasted hours). Percentile SLIs focus effort on the painful tail. Right-sizing avoids over-instrumentation cost (Module 21/Module 47).

Advantages / Disadvantages

- **+** Correlated production visibility (stability/usage/performance), safe monitored rollouts, fast diagnosis, data-driven iteration, verifies the whole app in the field, swappable/testable.
- **−** Setup + backend cost across pillars, correlation/consent/redaction discipline, alert tuning, SLO governance, right-sizing judgment.

Interview Questions

1. 🟢 **What makes an observability stack, not just tools?** — All three pillars (stability/usage/performance) correlated (shared ids + context), surfaced on dashboards, alerted, and wired into a feedback loop that drives action.
2. 🟢 **How do the pillars correlate?** — A shared correlation/trace id + context (version/device/session) links a crash to its breadcrumbs, funnel step, and performance trace — one incident view.
3. 🟡 **How does monitoring close the feedback loop?** — Signals gate staged rollout (promote/halt) → triage the correlated incident → forward-fix/hotfix → iterate on funnel/perf/stability data → re-verify.
4. 🟡 **Why route everything through the logging facade + interfaces?** — DRY, PII-safe/consent-gated instrumentation, swappable vendors, and testable (fakes) — one place owns redaction/consent/correlation.
5. 🟡 **What's on a release-health dashboard, and what alerts?** — Crash-free rate + key SLIs (p95 startup, jank%) + funnels per version; alerts on crash-free drop / SLO breach / conversion drop / error spike — actionable + tuned.
6. 🔴 **How does monitoring verify the rest of the handbook?** — Crash-free rate validates error handling/testing, SLIs validate performance budgets, funnels validate product decisions — the field verification loop of everything built.

7. 🔴 **How do you right-size the stack?** — Small: crash reporting + a few SLIs + basic analytics; large: full dashboards, tuned alerting, SLO governance, on-call, experiment analytics.

Senior Engineer Tips

- Correlate the pillars with one id + version/device context and feed them from your logging facade; the ability to jump from a crash to its funnel step and trace is what turns hours of guessing into minutes of diagnosis.
- Build the release-health dashboard + a handful of actionable alerts first, and wire them into the staged-rollout gate; monitoring only pays off when it drives the promote/halt/hotfix decision.
- Keep it PII-safe, consent-gated, low-overhead, and behind interfaces across all pillars — and right-size it; over-instrumenting a small app is as much a failure as flying blind on a large one.

Architect Perspective

Monitoring integration is the outer verification loop of the entire handbook: a correlated, privacy-safe, low-overhead observability stack (stability + usage + performance) that surfaces release health, alerts on real anomalies, and feeds the delivery loop (gate rollout → triage → fix → iterate). It confirms in the field that the architecture, performance budgets, and quality gates actually hold — closing build→ship→observe→improve. Assembled behind interfaces and fed by the logging facade, and right-sized to the product, it's what makes shipping a continuous, data-driven, safe operation rather than a leap of faith (01_monitoring_fundamentals.md, Module 51, Module 39, Module 47).

Summary

- Integrate all three pillars (crash/analytics/performance) behind interfaces, fed by the logging facade, correlated via shared ids + context; surface a release-health dashboard + actionable alerts.
- Close the feedback loop: gate staged rollout on signals → triage correlated incident → forward-fix/hotfix → iterate; verify architecture/perf/quality in the field.
- Keep PII-safe + consent-gated + low-overhead across pillars; right-size to the product; completes build→ship→observe→improve.

Revision Notes

- Three pillars integrated + correlated: crash reporting (crash-free rate/symbolication/breadcrumbs), analytics (taxonomy/funnels, privacy-safe), performance (startup/frame/network SLIs + SLOs). Correlate via id + context

(version/device/session/opaque user); feed from logging facade; behind interfaces (swappable/testable).

- Dashboards (release health per version) + actionable/tuned alerts (crash-free drop / SLO breach / conversion drop / spike). Feedback loop: staged-rollout gate → triage (correlated incident) → forward-fix/hotfix (flags/kill switch) → iterate → verify recovery.
- PII-safe + consent-gated + low-overhead (async/batched/sampled) all pillars; right-size; verifies whole handbook in field; closes build→ship→observe→improve.

Practice Questions

1. What turns three monitoring tools into an observability stack?
2. How does correlation speed up diagnosis, and how do you achieve it?
3. How does monitoring close the delivery feedback loop?

Coding Questions

1. Compose `CrashReporter` / `Analytics` / `PerformanceMonitor` interfaces fed by the logging facade, correlated by id + context.
2. Instrument one flow across all three pillars (crash context + funnel event + custom trace).
3. Define the release-health dashboard signals + alerts + the staged-rollout gate.

Mini Project

Observability stack (capstone — Flutter/monitoring): Integrate crash reporting (crash-free rate + symbolication + breadcrumbs), analytics (event taxonomy + a conversion funnel, privacy-safe), and performance monitoring (startup/frame/network SLIs + SLOs) behind interfaces, fed by your logging facade, correlated via a shared id + version/device/session context; build a release-health dashboard (crash-free + SLIs + funnel per version), set actionable alerts, and document the feedback loop (staged-rollout gate → triage correlated incident → forward-fix/hotfix → iterate). Acceptance: all three pillars integrated + correlated (id + context) + behind interfaces; fed by logging facade; PII-safe + consent-gated + low-overhead; release-health dashboard + tuned actionable alerts; documented feedback loop gating staged rollout; right-sized; verifies architecture/perf/quality in the field.