IJC
International Journal of Computer (IJC)
Peer-Reviewed International Journal
ISSN: 2307-4523
https://ijcjournal.org

A Classification of Post-Release Stability Monitoring Practices for Mobile Applications with a Large User Base

Mikita Burkouski
iOS Developer, EasyDacha Corp, Salt Lake City, USA
* *Corresponding author: nikita@easydacha.com | ORCID: 0009-0001-5723-5068*
Accepted: 15 October 2026 | Published: 20 November 2026

Abstract

A mobile release is not finished when a version reaches the store: users decide when to update, several versions stay in use, and failure signals arise on devices the release team does not control. Post-release observation is therefore a distinct engineering practice, yet research treats its objects separately, as problems of crash analysis, lifecycle testing, review mining or flaky tests. This review classifies post-release stability monitoring practices by the visibility mode of a failure, that is, by how the failure becomes observable once a version is in users' hands. The classification draws on a bounded literature corpus, platform documentation and the author's own release practice on a retail application with a large audience. Five classes are distinguished: self-announcing failures (crashes and hangs), silent failures (unexpected states; lifecycle and restoration failures), baseline-relative failures (regressions), instrument failures (analytics integrity) and gate-signal failures (flaky tests before release). A synthesis matrix crosses the classes with four release horizons. Audience scale makes only self-announcing failures visible early in a phased rollout; silent and instrument failures need instrumentation designed before release. A flaky test is treated by removing its cause, not by reruns.

Keywords: analytics integrity; crash reporting; flaky tests; mobile applications; phased rollout; post-release monitoring; silent failures

1. Introduction

A mobile release does not end when a new version is published. The version reaches users in stages, each user decides whether and when to update, and several versions of the same application remain in use at once [1], [2]. Failure signals arise on devices the team does not control, and many never reach it unless the application was built to send them. The classification proposed here draws on the author's own practice as an iOS engineer on a retail application with an audience of hundreds of thousands of downloads, where observation after each release was a routine stage of the release cycle, referred to below as the retail practice. It also draws on the build and release pipeline the author automated for a consumer application, referred to below as the pipeline practice.

Research covers the objects of this observation one at a time: crash reporting and bucketing at scale [3], [4], framework-specific exceptions [5] and the reproduction of field crashes [6]; graphical user interface (GUI) failures that leave the application running [7], resume and restart errors [8] and data loss on interruption [9]; user reviews after updates [10], [11]; and flaky tests in the pre-release pipeline [12]–[15]. Each line classifies causes or detection techniques within its own object. The closest cross-cutting survey categorises field-based testing techniques by the environment and the system under test [16], which leaves open which objects a release team has to watch after a version ships. Platform tooling groups stability metrics such as crash rate, unresponsive sessions, battery use and memory by what they measure [17], [18]; the moment a failure becomes visible lies outside that grouping. No prior classification of post-release monitoring practices for mobile applications by the way a stability failure becomes visible was identified in the reviewed literature.

The article makes three contributions. First, it classifies post-release stability monitoring into five classes by visibility mode, the way a failure becomes observable after a version reaches users. Self-announcing failures are those that produce a report by themselves; silent failures are those that leave the

application running and report nothing; baseline-relative failures are those that are wrong only against the previous version; instrument failures are those that corrupt the measurement itself; and gate-signal failures are those that corrupt the pre-release gate deciding what reaches users, above all flaky tests. Second, it crosses the classes with the release horizon, which is the sequence of four stages a version passes through: the pre-release gate is the set of automated checks before merge and release, the release candidate is the internal build checked last, the phased rollout is the exposure of the version to a growing share of users, and the steady state is the period in which several versions remain in use. The crossing forms a synthesis matrix whose cells carry a verdict and, where the literature provides one, a number. Third, it derives from the matrix a claim about audience scale: scale makes self-announcing failures visible early in a phased rollout but only multiplies the harm of silent and instrument failures, whose instrumentation therefore has to be designed before release.

2. Materials and Methods

2.1 Literature corpus

The corpus was assembled by a structured search of OpenAlex, which aggregates Crossref, PubMed and other indexes, restricted to computer science and to 2009–2026. Twelve queries were run: "mobile app stability monitoring", "mobile app crashes", "crash reporting", "Android lifecycle bugs", "app store reviews app updates", "flaky tests", "mobile app analytics", "mobile app release", "field failures mobile apps", "post-release defects mobile apps", "in-field telemetry mobile apps" and "mobile app monitoring review". The queries returned 236 records; 207 remained after the filters on year, abstract and document type, and 95 met the relevance threshold when screened against the inclusion criteria. The shortlist held 45 works: 30 from the screening and 15 known in advance from preliminary scoping searches, 9 of which the queries had not returned. The counts are taken from the search log and can be reproduced with the same queries; the procedure is a structured search and does not constitute a systematic review. Included works were peer-reviewed papers that studied failures of deployed mobile applications, mobile release and update practice, telemetry or data quality, or flaky tests, with empirical evidence from applications, users, tests or developers. Of the 45 works, 21 are cited; 11 of them were examined in full text and the others at the level of their reported results, so the placement of those studies in the classes carries some uncertainty. Four pages of platform documentation on phased and staged releases and on stability metrics bring the reference list to 25 sources.

2.2 The release practice as a second evidence class

The classes were drafted from the retail practice, on an application with more than 260,000 App Store downloads. Every release passed through task development, code review, testing, a release candidate, a final check of critical scenarios, publication and observation after release. After publication the author selected six objects to watch, looking for problems that appear only with real users: crashes, unexpected application states, lifecycle problems, restoration of screens after the application returned from background, regressions in scenarios that already worked and the correctness of analytics events after interface changes. In the pipeline practice, the same engineer separated pull-request checks, which gave a fast answer before merge, from a release workflow that signed and uploaded the build, and kept a test that failed at random out of the mandatory pipeline until its cause had been found and removed (hereafter, the gate rule). This operational material fixes the list of objects and the gate rule; it is not a measurement. Three limits follow from it and from the corpus: the applications are not named, no internal metrics are published, and the literature that populates the classes comes mostly from Android and partly from other platforms, so every piece of evidence is tagged with its platform.

2.3 Classification procedure

The classes were fixed before the search and the search populated them and tested each for emptiness. The basis of division is the visibility mode of a failure. A flat list of the watched objects was rejected because it has no basis of division; a division by signal source (crash reporter, platform metrics, analytics, reviews) was rejected because it repeats the groupings of platform tooling and sorts instruments, while the practices that use them cut across it; the signal source is kept as an attribute of each class. The release horizon is the same for every object, so it serves as the second axis of the synthesis matrix. Only stability failures were classified; performance, security and accessibility defects stay outside the classes.

3. Results

3.1 Visibility mode as the classification principle

The visibility mode of a failure is the way it becomes observable to the release team after a version reaches users. A failure can announce itself by producing a report, stay silent until an observer designed for it is in place, be visible only against the previous version, or corrupt the observer itself. The four modes give four post-release classes: self-announcing failures (K1), silent failures (K2), baseline-relative failures (K3) and instrument failures (K4). A fifth class, gate-signal failures (K5), covers the pre-release gate, whose signal decides what reaches users. Each class is named after how its failures become visible, and that mode fixes the monitoring practice the class requires: ranking reports for K1, observers designed before release for K2 and K4, a comparison with the previous version for K3 and a trustworthy gate signal for K5. Table 1 defines the classes.

Table 1. Classification of post-release stability monitoring by visibility mode

Class	How the failure becomes visible	Objects watched in the retail practice	Primary signal	Evidence (platform)	Borrowed sub-typology
K1 Self-announcing failures	The process ends or stops responding; a report arises without a dedicated observer	Crashes	Crash reporter; platform stability metrics	[3] desktop; [4] other mobile platform; [5], [6] Android; [18] iOS	–
K2a Unexpected application states	The application runs on in a wrong state; nothing is reported unless an event was designed for it	Unexpected application states	Designed state events; user complaints	[19] cross-platform; [7] Android	Missing and wrong GUI objects [7]
K2b Lifecycle and restoration failures	State is lost or corrupted on backgrounding, interruption or recreation; the loss shows on return, mostly without a crash	Lifecycle problems; restoration of screens after background	Designed restoration checks	[8], [9] Android	–
K3 Baseline-relative failures	Behaviour is wrong only in comparison with the previous version	Regressions in existing scenarios	Comparison with the previous version; update-level reviews	[10] iOS; [11] Android; [1], [2] both	–
K4 Instrument failures	The measurement fails, not the function: events are lost, duplicated or fire in the wrong place	Correctness of analytics after interface changes	Consistency checks of the event stream	[20] online experiments; [21] Android	Causes of sample ratio mismatch [20]
K5 Gate-signal failures (pre-release)	A test passes and fails on the same code, so a failed build stops carrying information	Flaky tests	Test outcome history	[12] Java open source; [13] Android; [14] mobile; [22] desktop browser	Root causes of flakiness [12], [13]

The release horizon is the second axis, with its stages labelled H0 (pre-release gate: pull-request checks and the mandatory pipeline), H1 (release candidate), H2 (phased rollout) and H3 (steady state). On iOS the phased rollout moves from 1 % of users with automatic updates on the first day to 100 % on the seventh and can be paused for up to 30 days [23]; on Android the developer raises the share over time or halts it [24]. Figure 1 marks the horizon at which each class first becomes visible.

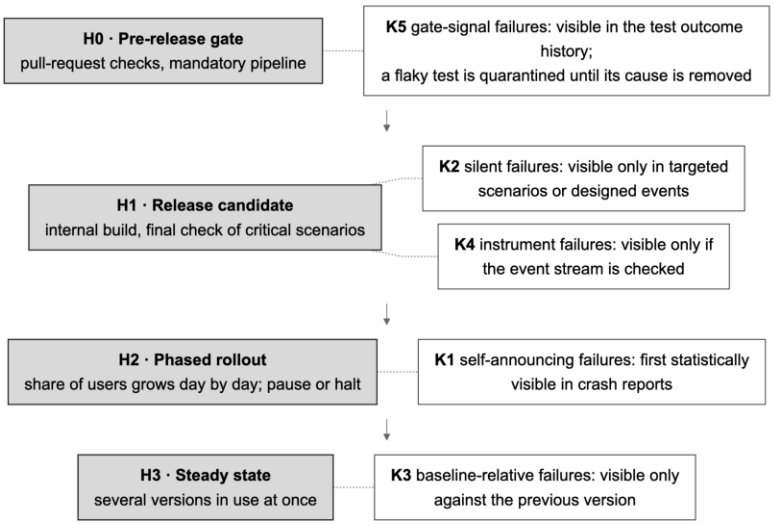


Figure 1. Release horizon and the point of first visibility of each class

The six objects of the retail practice fall into the classes without remainder: crashes into K1, unexpected states into K2a, lifecycle problems and restoration after background into K2b, regressions into K3 and analytics correctness into K4. A functional defect that does not crash the application lands in K2a when it corrupts the state on screen and in K3 when it breaks a scenario that worked in the previous version, so functional defects do not form a class of their own. A crash that a new version introduces into a scenario that worked before is both a crash and a regression; it belongs to K1, because the crash report makes it visible without any baseline, and the comparison with the previous version only dates it. Flaky tests enter as K5.

3.2 Self-announcing and silent failures (K1, K2)

Self-announcing failures need no observer to exist, only one to be counted. An error-reporting service collecting reports from a very large installed base groups them into buckets and uses the statistics of the buckets to isolate defects that could not be found at a smaller scale [3]. On one mobile platform, a study of 25 million crash reports found that over 90 % of the crashes came from 10 % of the root causes [4], so a few buckets carry most of the volume. On Android, exceptions raised inside the framework rather than in application code affected 75.3 % of the studied projects and made up 50.8 % of all exceptions [5]. Crashes that depend on the device context are hard to reproduce, and crowdsourced monitoring was proposed to recreate that context [6]. Hangs belong to the same class: Android vitals counts an application that stops responding against a threshold of 0.47 % of daily active users [17], and on iOS the platform delivers crash and hang diagnostics and daily metric reports from users' devices [18]. For K1 the open question after release is the ranking of buckets, because detection comes with the class.

Silent failures reverse this. Interviews with mobile developers found support for diagnosing unexpected application behaviour lacking across the major platforms [19]. For unexpected states (K2a), a study of orientation changes on Android found GUI failures in more than 88 % of the analysed applications, mostly missing or wrong interface objects in an application that kept running; in its open-source sample, 439 distinct failures appeared in 59 of 68 applications [7]. Such a failure is found by rotating the device in a targeted test, because in the field nothing reports it. For lifecycle and restoration failures (K2b), instance data that is not saved and restored between runs is lost or corrupted when the system pauses, backgrounds or kills the application [8], and in a study of data-loss faults only 38 of 298 affected activities produced crashes [9]. A crash reporter sees the small crashing part of this class; the rest leaves no trace unless the application was built to report a failed restoration.

3.3 Regressions against the previous version (K3)

A regression is a failure of a scenario that worked in the previous version, so it exists only relative to that baseline. Two properties of mobile distribution blur the baseline. Releases are frequent: one large industrial mobile deployment moved from a release every 8 weeks to a weekly release and exposed each Android release to 20 %, then 50 %, then 100 % of users [1]. Updates depend on users: of 654 surveyed users, 44.9 % always allowed automatic updates, and 47.7 % had encountered a problem after updating [2]. A regression therefore sits next to the unchanged behaviour of users who stayed on older versions, and it becomes visible only when the cohort of the new version is compared with the cohort of the previous one. User reviews carry the same signal with a delay. In low-rated reviews of 20 free iOS applications, 11 percent of the reviews attributed the complaint to a recent update [10], and an analysis of reviews per update, applied to the 250 worst updates in a large set of free applications, exposed bad updates that an application-level rating hides; feature removal and interface issues produced the sharpest rise in negative reviews, and bad updates with crashes or functional issues were the most likely to be fixed by a later update [11]. In the retail practice, the final check on the release candidate is the last comparison with the previous version before users see it.

3.4 When the instrument fails: analytics integrity (K4)

The sixth object of the retail practice, analytics correctness after interface changes, covers whether events are sent, whether they are duplicated and whether they fire in the right place. A failure of this kind leaves the function working and breaks the measurement. The user sees nothing, the crash reporter records nothing, and dashboards keep producing numbers; K4 is a class of its own because, unlike a K2 failure, it also blinds the observers that K2 depends on. No study of event integrity after interface changes was identified in the reviewed mobile literature. The nearest systematic treatment comes from online controlled experiments, where a sample ratio mismatch, a split of users between variants that departs from the designed ratio, serves as the symptom of broken data collection. Approximately 6 % of the experiments in one large experimentation programme exhibited such a mismatch; its authors distinguished 25 causes, grouped by the stage of the experiment at which they arise, and in one case lost telemetry left the analysis with 30 % fewer sessions than the control group [20]. That taxonomy was built for experiments and is borrowed here as a model of how an instrument fails. On the mobile side, a study of analytics used to improve Android applications found flaws in two widely used analytics tools, including differences of over 10:1 in the crash rates they calculated [21].

3.5 Trust in the gate signal: flaky tests (K5)

A flaky test passes and fails on the same code. Its damage lies in the gate: a red build that is sometimes noise teaches the team to rerun instead of reading the failure, and a real defect then passes the gate. The typology of causes is borrowed from Luo et al. [12], who classified the causes of 161 flaky-test fixes in open-source projects into ten categories; Async Wait, Concurrency and Test Order Dependency accounted for 77 % of them. The four causes met in the pipeline practice (asynchrony, dependency on the network, a race condition and state shared between tests) fall into four of these categories (Table 2). An Android study of 77 fixes in 29 projects found concurrency the leading cause, at 36 %, and added causes specific to the platform, such as dependence on hardware, system version or a third-party library [13]. Mobile developers perceive flaky tests as critical and tie many of them to interface elements [14], and a survey of the field treats flakiness as a threat to any method that assumes a test outcome depends only on the code [15].

Table 2. Flaky-test causes and remedies

Cause category (borrowed from [12])	As met in the pipeline practice	Remedy that removes the cause
Async Wait: 45 % of 161 fixes [12]	Asynchrony	Explicit synchronisation on the awaited event; 54 % of these cases were fixed this way [12]
Concurrency: 20 % [12]; 36 % of 77 Android fixes [13]	Race condition	Ordering or guarding access to shared data
Test Order Dependency: 12 % [12]	State shared between tests	Setting up or cleaning up the shared state; 74 % of these cases were fixed this way [12]
Network [12]	Dependency on the network	Isolating the test from the live network
Other categories of [12]: resource leak, time, input and output, randomness, floating-point operations, unordered collections	Not met as separate causes	Specific to each category
Android-specific causes: hardware, system version or library, 22 %; program logic, 12 % [13]	Not met as separate causes	Specific to each cause
Counter-practice: skipping or removing the test, 13 % of Android fixes [13]	Rejected	None: the covered behaviour loses its test

Luo et al. call the common approach of running a failed test several times and accepting any pass "rather unsatisfactory" [12], and 13 % of the Android fixes skipped or removed the test instead of fixing it [13]. The gate rule of the pipeline practice follows from both findings. A test that fails at random is quarantined, that is, it leaves the mandatory pipeline, and it returns only after its cause is removed; reruns do not count as a remedy. The rule keeps noise out of the gate signal and keeps the test, which deletion would lose. In the release history of a web browser, builds with flaky tests carried a median of 514 crash reports in beta and 234 in production, while builds with all tests passing had a median of only two [22]; in that history, a gate-signal failure ignored at H0 went together with more self-announcing failures after release.

3.6 Classes across the release horizon

Table 3 crosses the classes with the horizons and carries the article's synthesis.

Table 3. Classes across the release horizon: verdict and evidence

Class	H0 Pre-release gate	H1 Release candidate	H2 Phased rollout	H3 Steady state
K1 Self-announcing	Partial: only paths the tests exercise	Visible on a small internal population: in one industrial deployment, internal pre-release builds crashed about 10 times as often as production builds [1]	First statistically visible: exposure grows from 1 % to 100 % in 7 days, pause up to 30 days [23]; Android rollout can be halted [24]	Reliable as a trend: platform threshold 1.09 % of daily users with a crash, 8 % per device model [17]; per-user and per-session rates differ, 100 % against 33 % for one user [17]
K2 Silent	Only in targeted scenarios: orientation-change GUI failures in more than 88 % of analysed apps [7]	Visible if the final check includes background and return	Mostly invisible: only 38 of 298 activities with data loss crashed [9]	Late, through complaints, unless state events were designed [10], [19]
K3 Baseline-relative	Only for scenarios covered by regression tests	Last comparison with the previous version before users	Only against the previous cohort; per-update reviews expose bad updates [11]	Baseline blurred: 44.9 % of users always update automatically [2]; 11 percent of low-rated iOS reviews blame a recent update [10]
K4 Instrument	Invisible to functional tests	Visible if the event stream is checked after interface changes	Invisible to users and to crash reports; ratio anomalies in about 6 % of experiments of one programme [20]	Silent drift: two tools differed by over 10:1 in calculated crash rates [21]
K5 Gate signal	Visible as intermittent failures: three causes give 77 % of 161 fixes [12]; 13 % of Android fixes skipped the test [13]	Not applicable	Associated with more K1: median 514 beta crash reports for builds with flaky tests, two for passing builds [22]	Associated with more K1: median 234 production crash reports [22]

Along the rows, only K1 becomes visible by itself, and it does so first at H2, where the growing share of exposed users turns a rare crash into a count. K2 and K4 are visible before release only in targeted scenarios or event checks and after release only through observers designed in advance. K3 needs the previous version as a reference at every horizon, and K5 exists only at H0, while builds that ignored it carried more K1 reports at H2 and H3. Down the columns, H2 is the horizon at which most post-release signals first arrive, and H3 is the horizon at which they are hardest to compare, because versions and

measurement units multiply. The verdicts also locate the checks of the retail practice: the release candidate is the last horizon at which K2, K3 and K4 can all be examined before users see them, so restoration scenarios, the comparison with the previous version and the event stream belong in its final check. At H2 the crash reporter suffices for K1 alone; the other classes arrive there only as signals built at H1.

4. Discussion

4.1 What audience scale changes, class by class

Audience scale changes the visibility of K1 and leaves the other classes as silent as before. The number of users a failure affects in a day equals its rate among daily active users multiplied by the number of daily active users. The platform threshold of bad behaviour on Android puts the user-perceived crash rate at 1.09 % of daily active users [17]. The calculation that follows assumes a hypothetical audience of 100,000 daily active users, a round figure that describes no real application. A crash that affects 0.1 % of daily active users stays more than ten times below that threshold, yet it reaches 100 users every day. In a phased release the same multiplication runs over a growing base. On iOS the first day exposes 1 % of the users with automatic updates and the fifth day 20 % [23], so for 100,000 such users a crash affecting 1 % of exposed users reaches 10 users on the first day and 200 on the fifth. A large audience therefore makes a rare crash statistically visible within the first days of the rollout, while the rollout can still be paused; a small audience moves that point towards the end of the rollout or beyond it.

For K2 and K4 the multiplication raises the harm and leaves the signal where it was. A silent state loss that affects 0.1 % of daily users affects the same 100 users out of 100,000 each day, and if its crash share resembles the one observed for data-loss faults, most of them produce no report [9]. An analytics event lost after an interface change corrupts every decision taken on it, and a larger audience only makes the corrupted aggregate look more reliable. K3 sits between the two: scale makes the cohort of a new version large enough for comparison, but only when the comparison was set up in advance. The thesis that a large audience moves the centre of gravity of a release process from checking before release to observation after it therefore holds in full for K1 and holds for K2, K3 and K4 only when their observers are designed before release, as part of the release candidate. The comparison of rates also requires one unit. A user with three sessions and one crash in a day counts as a 100 % crash rate per user and 33 % per session [17], and two tools differed by over 10:1 in the crash rates they calculated [21].

4.2 Conditions under which the classification holds

The weight of each class shifts with the conditions of release. With a small audience, K1 stops being statistically visible early and moves towards H3, so observation relies more on the release candidate. Without a phased rollout, H2 collapses into H3: on Android a staged rollout is available only for updates [24], so the first publication of an application reaches all users at once. Distribution managed inside an organisation, where updates are installed centrally, shortens the tail of old versions at H3 and simplifies the baseline for K3. Performance degradation behaves as a baseline-relative object: a slower screen is a defect only against the previous version, and performance bugs of smartphone applications form a separate line of study [25]. Behaviour delivered as server-side configuration, such as a remotely switched feature, changes without a new version; the horizon then applies to the configuration change, and the baseline for K3 becomes the previous configuration. At H3 a defect fixed in the current version keeps producing K1 reports from users who have not updated, and with fewer than half of surveyed users always updating automatically [2], crash counts have to be read per version. Instrumentation can also move an object between classes: an unexpected state that the application reports as a handled event moves from K2 towards K1.

4.3 Open problems

Three problems follow from the thin cells of the matrix. The first (G1) is silent lifecycle failures on iOS. For iOS, no empirical study of K2b comparable to the Android ones [8], [9] was found, and the platform's diagnostics [18] cover crashes and hangs, not lost state. A lost state does not crash and so never enters crash datasets, yet returning from background is a routine user path. A first step is a catalogue of restoration failures on iOS built from targeted background-and-return scenarios, reporting the share that crashes next to the share that does not.

The second problem (G2) is analytics integrity after interface changes. Mobile work on analytics fidelity addresses how tools calculate crash rates [21], and the systematic treatment of broken measurement comes from online experiments [20]. A broken event produces no visible failure, while decisions about a new version rest on it. A first step is a set of event-stream invariants checked on the release candidate, for example that each screen emits its view event once and each action emits its event after the action completes.

The third problem (G3) is the baseline for regressions under user-controlled updates. Users control their updates [2] and reviews tie complaints to updates [10], [11], but no description of the practice of comparing the cohort of a new version with the cohort of the previous one was identified in the reviewed literature. Server-side experimentation assumes a controlled split of users, which a mobile update does not provide, and early updaters select themselves. A first step is a description of version-cohort baselines that account for that self-selection.

4.4 Boundaries of applicability

The classes cover stability failures only; performance, security and accessibility defects were not classified. The classification rests on a bounded literature corpus, of which 11 studies were examined in full text and the rest at the level of their reported results, and on a release practice described in words, without named applications or internal metrics. Most empirical evidence in the classes comes from Android and other platforms, which is why each piece of evidence carries a platform tag; the iOS evidence rests on one study of user reviews and on platform documentation.

5. Conclusion

After a mobile release, stability is watched through more than crashes. The objects of post-release observation differ in how their failures become visible: crashes and hangs announce themselves, unexpected states and lifecycle failures stay silent, regressions exist only against the previous version, analytics failures corrupt the observer, and flaky tests corrupt the pre-release gate that decides what reaches users. Crossing these classes with the release horizon shows where each failure can first be seen and by which signal. Audience scale turns a rare crash into a count within the first days of a phased rollout; it does not do the same for a silent or an instrument failure, whose observers have to exist before release. A flaky test is treated by removing its cause, and until then it stays out of the mandatory pipeline. The release candidate is where a post-release practice is built: its final check is the place to prepare the event streams, the restoration paths and the comparison with the previous version that observation after release will rely on. The visibility mode of a failure, not its frequency, determines what a release team has to build before shipping and what it can leave to its audience to reveal.

Acknowledgements

None.

References

[1] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm, "Continuous deployment of mobile software at facebook (showcase)," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*, 2016, pp. 12–23, doi: 10.1145/2950290.2994157.

[2] M. Nayebi, B. Adams, and G. Ruhe, "Release Practices for Mobile Apps – What do Users and Developers Think?," in *Proc. IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 552–562, doi: 10.1109/saner.2016.116.

[3] K. Glerum et al., "Debugging in the (very) large," in *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles (SOSP)*, 2009, pp. 103–116, doi: 10.1145/1629575.1629586.

[4] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proc. 12th Annu. Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, 2014, pp. 190–203, doi: 10.1145/2594368.2594377.

[5] L. Fan et al., "Large-scale analysis of framework-specific exceptions in Android apps," in *Proc. 40th Int. Conf. Software Engineering (ICSE)*, 2018, pp. 408–419, doi: 10.1145/3180155.3180222.

[6] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *Proc. Int. Conf. Mobile Software Engineering and Systems (MOBILESoft)*, 2016, pp. 88–99, doi: 10.1145/2897073.2897088.

[7] D. Amalfitano, V. Riccio, A. C. R. Paiva, and A. R. Fasolino, "Why does the orientation change mess up my Android application? From GUI failures to code faults," *Software Testing, Verification and Reliability*, vol. 28, no. 1, Art. no. e1654, Jan. 2018, doi: 10.1002/stvr.1654.

[8] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in Android applications," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, pp. 864–880, doi: 10.1145/2983990.2984011.

[9] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, "Data loss detector: automatically revealing data loss bugs in Android apps," in *Proc. 29th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2020, pp. 141–152, doi: 10.1145/3395363.3397379.

[10] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What Do Mobile App Users Complain About?," *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015, doi: 10.1109/ms.2014.50.

[11] S. Hassan, C.-P. Bezemer, and A. E. Hassan, "Studying Bad Updates of Top Free-to-Download Apps in the Google Play Store," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 773–793, Jul. 2020, doi: 10.1109/tse.2018.2869395.

[12] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*, 2014, pp. 643–653, doi: 10.1145/2635868.2635920.

[13] S. Thorve, C. Sreshtha, and N. Meng, "An Empirical Study of Flaky Tests in Android Apps," in *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, 2018, pp. 534–538, doi: 10.1109/icsme.2018.00062.

[14] V. Pontillo, F. Palomba, and F. Ferrucci, "Test Code Flakiness in Mobile Apps: The Developer's Perspective," *Information and Software Technology*, vol. 168, Art. no. 107394, Apr. 2024, doi: 10.1016/j.infsof.2023.107394.

[15] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–74, Jan. 2022, doi: 10.1145/3476105.

[16] A. Bertolino et al., "A Survey of Field-based Testing Techniques," *ACM Computing Surveys*, vol. 54, no. 5, pp. 1–39, Jun. 2022, doi: 10.1145/3447240.

[17] "Android vitals," Android Developers. [Online]. Available: https://developer.android.com/topic/performance/vitals (accessed Sep. 25, 2026).

[18] "MetricKit," Apple Developer Documentation. [Online]. Available: https://developer.apple.com/documentation/metrickit (accessed Sep. 25, 2026).

[19] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl, "Diagnosing mobile applications in the wild," in *Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010, pp. 1–6, doi: 10.1145/1868447.1868469.

[20] A. Fabijan et al., "Diagnosing Sample Ratio Mismatch in Online Controlled Experiments," in *Proc. 25th ACM SIGKDD Int. Conf. Knowledge Discovery & Data Mining (KDD)*, 2019, pp. 2156–2164, doi: 10.1145/3292500.3330722.

[21] J. Harty, "Improving app quality despite flawed mobile analytics," in *Proc. IEEE/ACM 7th Int. Conf. Mobile Software Engineering and Systems (MOBILESoft)*, 2020, pp. 21–22, doi: 10.1145/3387905.3388603.

[22] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in *Proc. 26th ACM Joint Meeting European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 857–862, doi: 10.1145/3236024.3275529.

[23] "Release a version update in phases," App Store Connect Help. [Online]. Available: https://developer.apple.com/help/app-store-connect/update-your-app/release-a-version-update-in-phases (accessed Sep. 25, 2026).

[24] "Release app updates with staged rollouts," Play Console Help. [Online]. Available: https://support.google.com/googleplay/android-developer/answer/6346149 (accessed Sep. 25, 2026).

[25] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. 36th Int. Conf. Software Engineering (ICSE)*, 2014, pp. 1013–1024, doi: 10.1145/2568225.2568229.