One Rule, Three Runtimes: Differential Testing of a Daily-Content Rotation Across Dart, Swift and Kotlin

Yousof Selim
Sunway University (Lancaster University dual degree), Malaysia

0% of 20 real and seeded bugs caught by string-matching tests

1. All three copies agreed on 95.66% of 550,420 test instants, chosen to cluster around clock changes in 29 time zones.
2. The disagreements trace to eight divergences: seven around skipped or repeated hours, and one sub-second rounding of the schedule anchor.
3. Worst case, in the Swift code run on macOS: if a widget's three-month schedule runs out on a midnight clock change, it shows one quote behind every day until the app is opened.
4. The 22 string-matching tests caught 0 of 12 hand-written behaviour-changing mutants; running the code caught all 12.

Limits. One rule in one app, chosen inputs rather than real usage, hand-written mutants, and no phones: Kotlin ran on a desktop JVM and Swift on macOS.

Abstract

The problem Adelante's widgets change quotes while the app is closed, so the rule that decides which quote is showing exists three times: in Dart, Swift and Kotlin. The only guard was 22 tests that search the native source for certain strings.

What I did I ran copies of all three implementations side by side on 550,420 test instants in 29 time zones, chosen to stress clock changes,

	midnight transitions, leap days and long horizons, and compared the quote each would show.
What I found	**They agreed on 95.66%. The rest trace to eight divergences, seven of them around skipped or repeated hours, and none made a string test fail. On 12 hand-written faults, the string tests caught none and running the code caught all 12.**
Why it matters	Copies of shared logic drift where each language computes local time itself. Publishing change instants further ahead and testing what the code computes, not how it is spelled, is cheap and catches what string checks cannot. Swift ran on macOS and Kotlin on a desktop JVM, so device confirmation is still needed.

Contributions

- A differential harness that runs copies of the Dart, Swift and Kotlin versions of one shipping rule on 7,658 synthetic publications in 29 time zones.
- Eight reproduced divergences, each with its root cause, trigger, user-visible effect, duration and likelihood, plus seven latent ones in payloads the app does not yet publish.
- A comparison on 14 hand-written mutants: string-matching tests caught 0 of 12 behaviour changes and failed on 2 of 2 harmless refactors; the differential test caught 12 of 12 and flagged neither refactor.
- Practical guidance for logic shared across native targets: publish instants further ahead, keep one notion of a day, and test behaviour rather than text.

Keywords. differential testing · cross-platform consistency · time zones · daylight saving time · Flutter · WidgetKit · Android app widgets · mutation analysis · contract tests

1 Introduction

Adelante is a quotes app I built in Flutter. It has been on the App Store since 4 October 2026. Its home-screen widgets are not Flutter: on iOS they are

WidgetKit extensions in Swift, and on Android they are app widgets in Kotlin. A widget has to change its quote on schedule while the app is closed, so the rule that decides which quote is showing at a given moment exists three times: in Dart, in Swift and in Kotlin.

The three copies must agree. When the user opens the app, the Dart side works out which quote should be current and republishes the widget. If a native copy had reached a different answer, the widget visibly jumps to another quote. Nothing crashes and no test fails. It just looks wrong.

The project's guard for this was a set of 22 structural contract tests. The Dart test runner cannot compile Swift or Kotlin, so these tests read the native source files as text and check that certain strings are present, such as `currentPoolIndex` or `payload.interval > 0`. All 22 pass on the shipping code.

This report asks whether the three copies actually compute the same thing. I answer with differential testing [1]: I lifted each implementation into a runnable harness, gave all three the same generated inputs and compared their answers instant by instant. Running several versions side by side is also the core of N-version programming [2], and Knight and Leveson showed that such versions fail together more often than independence would predict [3]. Here the versions were not written independently. Each was written to mirror the others, which makes a disagreement more telling when it appears.

Research questions

Table 1 lists the four questions and where each is answered.

Table 1. Research questions and the experiments that answer them.

Question	Experiment	Answered in
RQ1. How often do the three copies agree, and where do they disagree?	Differential run: 550,420 instants, 29 zones	Section 5.1
RQ2. What causes each disagreement, what would a user see, and how likely is it?	Mechanical classification, library probes, hand-built payloads	Sections 5.2 to 5.4
RQ3. Do the string-matching contract tests detect these divergences, and how do they compare with a differential test on seeded faults?	Contract tests on shipping code; 14 single-line mutants	Section 5.5
RQ4. What does the differential test cost to run?	CPU time of the full run	Section 5.6

In short: the copies mostly agree, but not always. All three gave the same answer on 95.66% of the test instants, which were deliberately concentrated around clock changes. The disagreements trace to eight distinct divergences. Seven concern what "the same local time tomorrow" means when a clock change removes or repeats an hour, a known and common class of date and time bug [4]. The eighth is a sub-second truncation. None of the eight made any of the 22 contract tests fail.

Contributions

- A differential harness for one shipping rule. It runs copies of the Dart, Swift and Kotlin functions on 7,658 synthetic widget publications in 29 time zones.
- Eight reproduced divergences, each with its root cause, trigger, user-visible effect, duration and likelihood. Seven more appear only for payloads the current app never produces.
- A comparison of oracles on 14 hand-written mutants. The 22 string-matching tests caught 0 of 12 behaviour-changing mutants and failed on 2 of 2 harmless refactors. The differential test caught 12 of 12 and flagged neither refactor.
- Practical guidance for logic shared across native targets: publish instants further ahead instead of computing past the list, and test executable behaviour rather than spelling.

2 Background and related work

Differential testing and N-version programming

McKeeman described differential testing as giving the same input to several comparable systems and treating any difference in output as a candidate bug [1]. Its strength is that it needs no specification of the correct answer. Its weakness is the mirror image: if every system makes the same mistake, the test is silent. Csmith is the best-known large-scale example. It generated random C programs, compared compilers on them and found hundreds of compiler bugs [5].

N-version programming uses the same comparison at run time to tolerate faults [2]. Knight and Leveson's experiment with 27 versions found that independently developed programs failed together more often than chance would predict [3]. Programmers find the same inputs hard, so shared difficulty produces shared mistakes.

Adelante's situation is the reverse of an N-version system. The three copies are deliberately dependent: the native comments say they "mirror" the Dart function. Differential testing still applies, but it can only show where the copies disagree. It cannot show that an answer they share is the one the user wanted.

Metamorphic and property-based testing

Metamorphic testing checks relations between the outputs for related inputs, for example "moving every instant by one rotation period advances the index by one" [6] [7]. Property-based testing, popularised by QuickCheck, generates many inputs and checks a stated property on each [8]. Both suit this problem. I chose plain differential testing because the three implementations already provide an oracle. The vector generator borrows from property-based testing: it is seeded and deliberately biased towards edge cases.

Time-zone and date bugs

The IANA time-zone database records each zone's rules and their history [9]. Two facts from it drive this study. First, local clock times can be skipped (a gap, when clocks go forward) or repeated (an overlap, when they go back). Second, zones disagree on when those changes happen. Most happen between 01:00 and 03:00, some at midnight, and Lord Howe Island moves by only 30 minutes. Each language's calendar library has to decide what a skipped or repeated time means. Tiwari et al. studied 151 date and time bugs in open-source Python projects and found time-zone mistakes to be the largest

contributing factor, with root causes often in misunderstood library behaviour such as default conventions [4]. My results show the same pattern across languages.

Cross-platform consistency in mobile apps

Fazzini and Orso's DiffDroid compares one Android app's behaviour across many devices and platform versions to find cross-platform inconsistencies [10]. My setting differs in two ways. The implementations are in different languages rather than one binary on different platforms, and the comparison is of a pure function rather than of user interfaces. The aim is the same: one app, one intended behaviour, several runtimes.

Mutation analysis

Mutation analysis judges a test suite by seeding small syntactic changes into the code and counting how many it detects [11]. Mutant detection correlates with real-fault detection in some settings [12], but that evidence comes from large, systematically generated mutant sets. I use 14 hand-written mutants only to illustrate how two oracles differ, not to estimate a detection rate.

Positioning

Differential testing is usually applied to independent implementations of a shared specification, such as compilers or virtual machines. This report applies it to something smaller and more common in mobile development: copies of one function that a single developer maintains in three languages because the platform forces it. It also compares the result with the string-matching tests that projects often use when their main test runner cannot execute the other languages. I do not claim the technique is new. The contribution is a measured case study of where such copies drift, and of what the cheaper test misses.

---

3 The rotation rule and its three implementations

A widget configuration has a pool of up to 96 quotes, chosen and ordered by the Dart app. The quote that is current at publication comes first. The user picks a cadence: manual, hourly, twice daily, daily or weekly. A daily cadence can also have a local clock time, such as 07:00. The rule answers one question: at instant t, which pool index is showing?

What Dart publishes

When the app publishes, `widget_payload.dart` writes a small record for the native side. For fixed intervals it writes an anchor epoch in whole seconds and the interval length in seconds. For a daily clock time it also writes the next 96 local change instants, computed in Dart with the device's time zone and converted to epoch seconds. The comment explains why: so the extensions "need no timezone database and never assume that a day lasts 24 hours". After those 96 instants, roughly three months, the native code has to extend the schedule itself. A flag, `initialQuoteOnly`, marks a quote the user chose by hand for Today. That quote holds only the first period and is not repeated when the pool wraps.

Three answers in Dart, Swift and Kotlin

Dart, the app. When the app opens it does not call a single "index at time" function. It reconciles its saved queue in `providers.dart`. It counts elapsed changes with `elapsedDailyChanges` (daily clock time) or `elapsedWidgetRefreshPeriods` (fixed intervals) and shows `queue[elapsed % n]`. With a Today override, it handles the first period separately and then counts the rest from `nextRefreshAt + interval`.

Dart, `currentPoolIndex`. A pure function in `widget_rotation.dart` computes `floor((t − anchor) / interval) mod n`. Both native files say they mirror it. In the shipped code it is called only from tests, never from `lib/`. I include it as a fourth answer because it is the documented rule.

Swift. `AdelantePayload.scheduledQuote` counts the published instants up to t with `prefix { $0 <= date }`. Past the last one, `dailyStep` adds whole calendar days using `Calendar.current`, `startOfDay` and `date(bySettingHour:minute:second:of:)`. WidgetKit does not run this function continuously. `adelanteTimeline` precomputes up to 12 future entries, using `Calendar.nextDate(…, matchingPolicy: .nextTime)` once the published instants run out, and the system displays the latest entry whose date has passed.

Kotlin. `scheduledIndex` counts the published instants with `count { it <= now }`. Past the last one it adds civil days and compares against `dailyBoundary`, which uses a lenient `java.util.Calendar` with the hour and minute fields set. An inexact `AlarmManager` window scheduled at `nextChange()` triggers each re-render.

Table 2 lists the concerns each copy has to get right. The three answers agree by construction only where the Dart-computed instants are used directly. Everywhere else, each language makes its own choices.

Table 2. Where each implementation does its own calendar arithmetic.

Concern	Dart app	Swift widget	Kotlin widget
Fixed interval	milliseconds from `nextRefreshAt`	seconds (Double) from anchor	whole seconds from anchor
Daily, first 96 changes	own day count	count of published instants	count of published instants
Daily, after 96 changes	own day count	`startOfDay` + `bySettingHour`	UTC civil day + lenient `Calendar`
When the display updates	on open	12-entry timeline, `nextDate(.nextTime)`	alarm at `nextChange()`
Today override	`next + Duration(days: 1)`	skip slot 0 after first step	skip slot 0 after first step

4 Method

4.1 Extraction

I copied the relevant source files from Adelante's repository at commit `af1aea2`, the 1.0.0 (21) code, into a scratch directory. I recorded SHA-256 hashes and did not modify the repository. The rotation files are clean in that commit. Each harness contains the rotation functions copied character for character, marked `VERBATIM` with their source line numbers. For Swift, a script checked that every non-comment line of the copied ranges appears in the harness. That check needs the private source, so it cannot be rerun from the artifact. The changes I had to make are mechanical and listed in the artifact:

- SwiftUI and WidgetKit types are removed. `Timeline` becomes a plain struct with the same entries and refresh date.
- Android's `SharedPreferences` is replaced by a small interface backed by the configuration-catalog JSON. It reproduces the override in `WidgetConfigurations.resolve`, which is how the shipping widget reads its data.
- Two Kotlin functions lose their `private` modifier.

The Dart answer for the app is the arithmetic extracted from `providers.dart`, not the provider itself, which depends on Flutter and Riverpod.

4.2 Test inputs

A seeded Python generator (seed 20261102) proposes inputs only. It never computes an answer. The inputs are chosen, not sampled from real use: no usage data was collected. The generator covers 29 zones picked for awkward rules:

- no clock changes under current rules (11): UTC, Kuala Lumpur, Singapore, Kolkata, Kathmandu at +05:45, Kiritimati at +14, Moscow, Asunción, São Paulo, Tehran and Apia (Samoa);
- changes in the small hours (10): New York, Los Angeles, St John's, London, Dublin (with its negative-DST encoding), Berlin, Gaza, Sydney, Auckland, and Chatham at +12:45 changing at 02:45;
- changes at midnight (5): Santiago, Havana, Cairo, Beirut and the Azores;
- unusual changes (3): Nuuk, which skips 23:00 to 00:00, Lord Howe's 30-minute change, and Troll's two-hour change.

For each zone the generator finds every UTC offset change from 2026 to 2030 by bisection. To stress historical rules it also adds Samoa's changes in 2011 and 2012, including the skipped 30 December 2011, and the changes in Cairo, São Paulo and Tehran from 2017 to 2023. Around each change it builds five families of publications (Table 3):

- daily publications made one to three days before a clock change;
- daily-tail publications made about 96 days before one, so that the published instants run out across it;
- interval publications with millisecond-precise publish times;
- long horizons of up to 13 years, with queries past 19 January 2038;
- leap-day publications in 2028 and 2032.

Daily clock times include 00:00, 00:01, 00:30, 00:59, 01:00, 01:30, 02:00, 02:30, 03:00, 09:00, 23:00, 23:30 and 23:59. Pool sizes are 1, 2, 3, 5, 7, 24, 95 and 96. Each publication outside the leap family carries a Today override with probability 0.35; 2,606 of the 7,658 do. Query instants sit at each local change minute (both readings of a repeated time), one millisecond either side, around each offset change, and at random offsets.

Table 3. Input families.

Family	Publications	Query instants
daily (inside the 96 published changes)	2,756	174,959
daily-tail (after the 96 changes)	2,756	230,145
interval (hourly, 12 h, 24 h, weekly, manual)	1,740	135,630
long horizon	232	1,856
leap day	174	7,830
Total	**7,658**	**550,420**

The queries are deliberately weighted towards boundaries. An agreement rate here is therefore a rate over hard cases, not over a typical day, and a uniform sample of instants would show far higher agreement.

4.3 Harnesses and pipeline

Figure 1 shows the pipeline. For each zone, the Dart harness runs with `TZ` set to that zone. It reproduces the publication: `nextDailyChange` or `t0 + interval`, the anchor and the 96 instants. It then records two answers per query: the app's reconciliation and `currentPoolIndex`. The Swift and Kotlin harnesses run in the same zone, read Dart's published payloads and record two answers each. The first is the instantaneous rule at t. The second is the displayed quote: the latest WidgetKit timeline entry, or the latest alarm-driven render, at or before t. The Swift and Kotlin harnesses record the zone their runtime reports, and the analysis stops if it differs from the requested one.

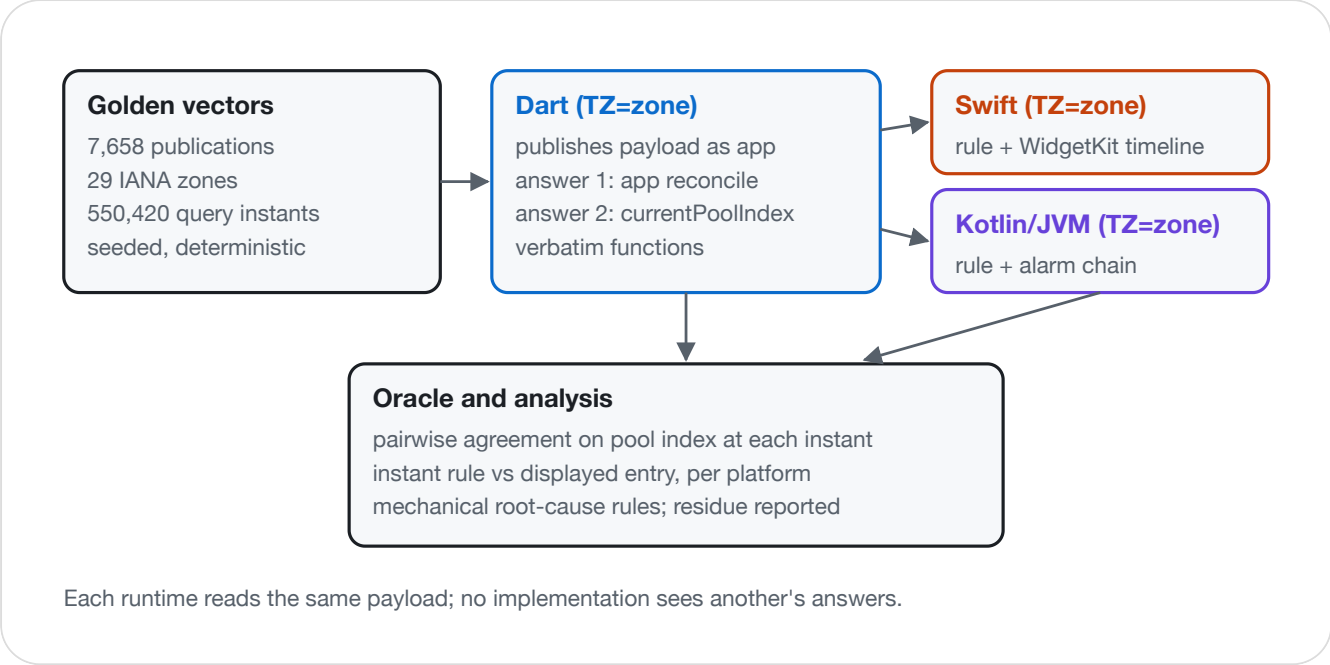


Figure 1. The differential pipeline. Each runtime runs in its own process with the same TZ.

The displayed answers rest on two idealisations. WidgetKit is assumed to request a new timeline exactly at its refresh date, and Android's ten-minute alarm window is assumed to fire at its start. Real devices can only be later, so these figures understate staleness.

4.4 Oracle and classification

There is no independent ground truth. I take the Dart app's reconciliation as the reference, because it is what the user sees after opening the app and what the widget is republished to. Any native answer that differs from it is a visible jump.

Each disagreement is assigned a class by fixed rules on three things: which runtime deviates, the publication type, and the local calendar facts at the query (whether the change minute is skipped or repeated that day, and whether the last published instant falls on a day that starts after midnight). Two residual labels catch anything the rules do not explain. The rules assign classes by pattern; they do not prove the cause of each individual query. I therefore reproduced one example of each class with a small probe of the library call involved.

4.5 Comparing with the string-matching tests

I ran the real 22 contract tests with Flutter 3.44.1 on a scratch copy of the files they read. I then wrote 14 single-line mutants of the native rotation code by hand [11]: 12 that change behaviour and 2 refactors that do not. Each mutant was applied identically to the contract-test copy and to the harness copy. A mutant counts as detected by the contract tests if

any of the 22 fails. It counts as detected by the differential test if any answer changes over the 220,228 query instants in six zones (UTC, New York, London, Lord Howe, Santiago and Kuala Lumpur).

4.6 Toolchains

Dart 3.12.1 (Flutter 3.44.1); Swift 6.4 (swiftlang-6.4.0.34.1) with Xcode 27.1 on macOS 27.0.1, Apple M1 Pro, Foundation time-zone data 2026e; Kotlin 2.4.21 on OpenJDK 17.0.19, JVM tzdb 2026a; Python 3.14.2 for generation and analysis. Adelante's Android build uses the Kotlin Gradle plugin 2.3.20. Kotlin 2.4.21 was installed from Homebrew because no standalone compiler was present. No phones, simulators or emulators were used.

4.7 Replication

The artifact contains the generator, the three harnesses, the run, analysis, mutant and figure scripts, the generated inputs and the raw per-zone answers. Rerunning the analysis on the published raw answers reproduces every agreement rate and class count in Section 5 exactly. Rerunning the harnesses needs the toolchains above. The answers are deterministic for given inputs and time-zone data, so a different tzdb version is the most likely source of differences.

---

5 Results

5.1 Agreement (RQ1)

Across all 550,420 queries, the three runtimes gave the same pool index for 526,530 (95.66%). Swift and Kotlin agreed with each other on 546,248 (99.24%). Table 4 breaks this down by input family. Leap days caused no disagreement at all, and neither did the zones without clock changes, apart from the sub-second effect D1.

Table 4. Agreement rate by input family (boundary-weighted queries).

Family	All three	App = Swift	App = Kotlin	Swift = Kotlin	`currentPoolIndex` = app
interval	86.34%	86.34%	86.34%	100.00%	86.34%
daily	99.44%	99.44%	99.44%	100.00%	89.54%
daily-tail	98.10%	98.47%	99.53%	98.19%	90.49%
long	99.84%	99.84%	99.84%	100.00%	99.62%
leap	100.00%	100.00%	100.00%	100.00%	97.25%
All	**95.66%**	95.82%	96.26%	99.24%	89.29%

These are exact counts over a fixed, chosen input set, not estimates from a random sample, so a confidence interval would not mean anything here. They describe how often the copies disagree on hard cases, not how often a user sees a wrong quote.

The displayed quote matched the instantaneous rule on 99.989% of queries for Swift and 99.948% for Kotlin. The rules correctly handle leap days, the 2038 limit of 32-bit epochs, and offsets such as +05:45 and +12:45. What they do not handle consistently is local clock times that are skipped or repeated.

The classification rules assigned every disagreement to a named class; both residual labels stayed empty. Figure 2 summarises the eight classes reachable with payloads the current app publishes. The counts include the historical transitions added in Section 4.2.

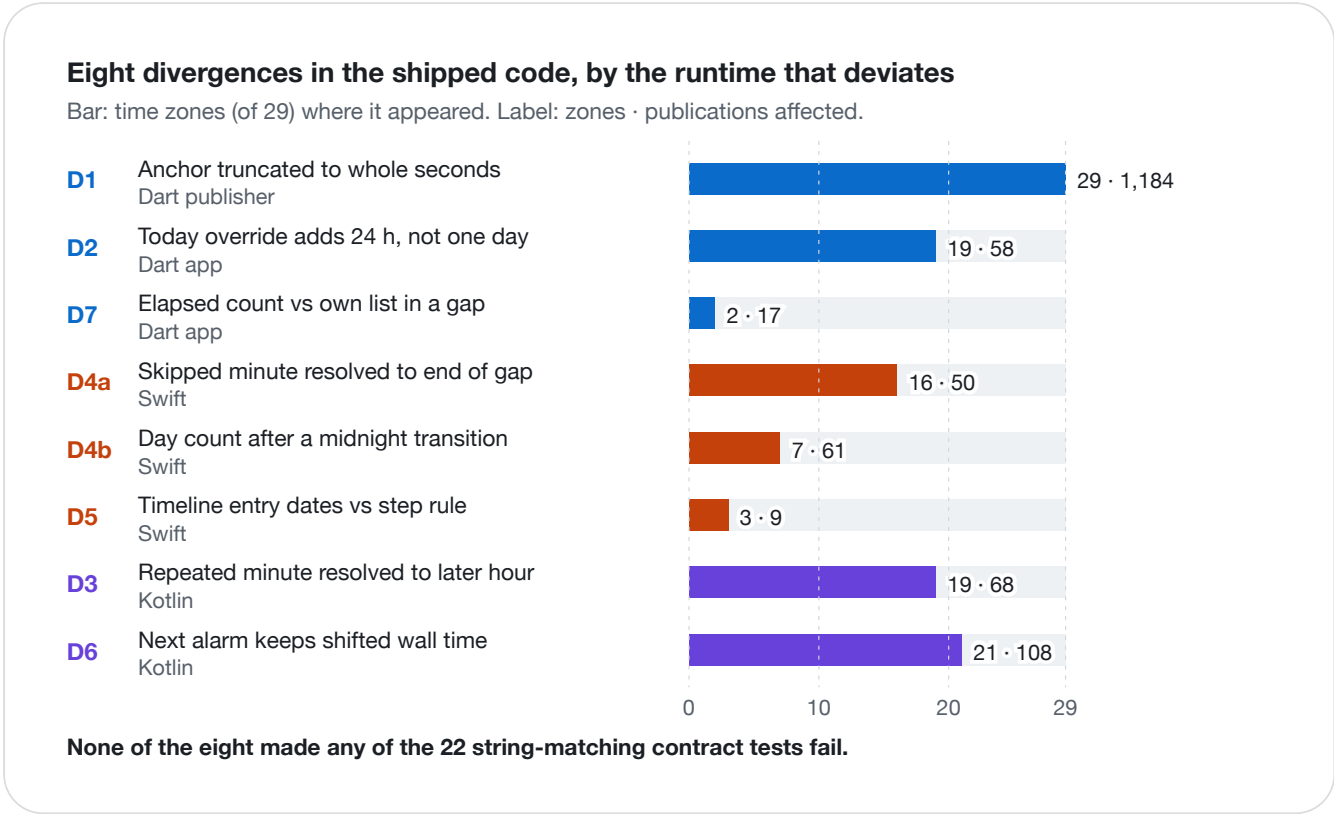


Figure 2. The eight divergences reachable with current payloads, coloured by the runtime that deviates: Dart blue, Swift orange, Kotlin purple. Bars show time zones; labels show zones and publications.

5.2 The eight divergences (RQ2)

Below, "tail" means a daily widget whose 96 published change instants have run out. That happens when the app has not been opened for about three months, which is plausible for someone who uses only the widget. Each divergence is described with its trigger, what the user would see, and how long it lasts. Table 5 then summarises their impact and likelihood.

D1. Anchor truncated to whole seconds (Dart publisher; 29 zones, 1,184 publications). The app's next change is `selectedAt + interval`, kept to the millisecond. The payload writes the anchor as `millisecondsSinceEpoch ~/ 1000`, so both native copies change the quote up to 999 ms before the app would. D1 accounts for every disagreement in the interval family (18,533 queries) and one in the long family; the generator deliberately probes the last second before each boundary. No user would notice it.

D2. Today override adds 24 hours, not one day (Dart app; 19 zones, 58 publications). With a Today override, the app counts later periods from `nextRefreshAt.add(refreshMode.interval)`, which is `Duration(days: 1)`: exactly 24 hours. Across a clock change that instant lands an hour away from the true next change, and `elapsedDailyChanges` miscounts. Example: New York, a change set for 03:00,

override chosen on 10 March 2028. At 03:00 EDT on 12 March both widgets have moved to pool index 2. The app, if opened then, computes index 1 and republishes, so the widget steps back a quote. Usually the window is one hour. When the shifted instant also crosses midnight, the app stays one off for the rest of that publication, as for a 23:59 change in St John's from 11 March 2029, or a 00:01 change in Nuuk when clocks went back on the night of 27 to 28 October 2029. It needs a Today override and a clock change within a day of the first scheduled change.

D3. A repeated minute resolved to the later hour (Kotlin tail; 19 zones, 68 publications). On a fall-back day, 01:30 happens twice. Dart and Foundation resolve `01:30` to the first occurrence (daylight time). A lenient `java.util.Calendar` with `HOUR_OF_DAY` and `MINUTE` set resolves it to the second (standard time). Kotlin's tail therefore changes the quote one hour late: in New York on 1 November 2026, from 01:30 EDT until 01:30 EST (Figure 3, top). This was observed on a desktop JVM. Android's `Calendar` implementation differs, so D3 needs confirming on a device.

D4a. A skipped minute resolved to the end of the gap (Swift tail; 16 zones, 50 publications). On a spring-forward day, 02:30 does not exist. Dart's `DateTime` and Java's lenient calendar both move it forward by the length of the gap, to 03:30. Foundation's `date(bySettingHour: 2, minute: 30, …)` returns 03:00, the first instant after the gap. Swift's tail therefore changes the quote 30 minutes early in Los Angeles on 14 March 2027 (Figure 3, bottom). The early window equals how far into the gap the change time falls; a change set exactly at the start of a gap agrees.

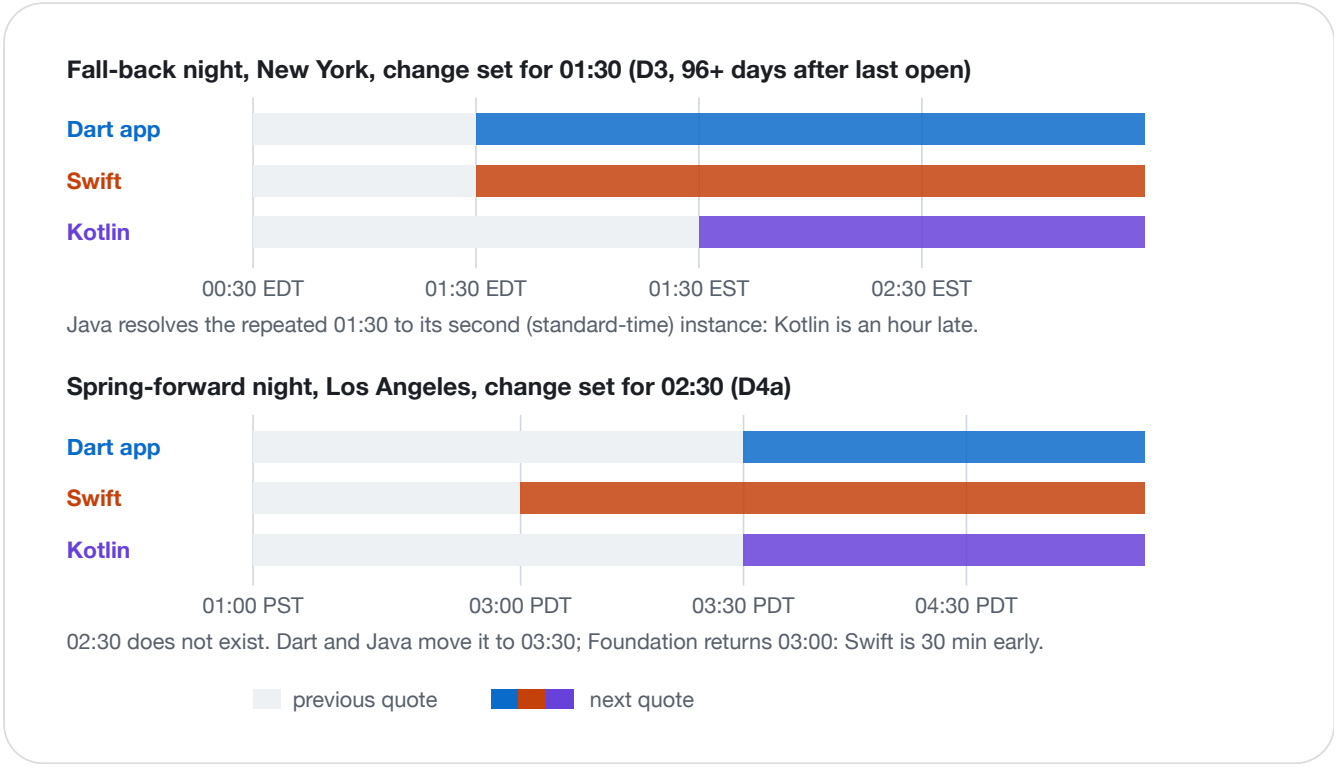


Figure 3. When each runtime switches to the next quote on two clock-change nights. Schematic, drawn from measured examples (publications s00648 and s00997); not to scale.

D4b. Day count after a midnight change (Swift tail; 7 zones, 61 publications). This is the most consequential divergence. In zones where clocks go forward at midnight, such as Santiago, Havana, Cairo, Beirut and the Azores, the day of the change begins at 01:00. If the last published instant falls on such a day, Swift computes `dateComponents([.day], from: startOfDay(last), to: startOfDay(date))`. From 01:00 on that day to 00:00 the next is 23 hours, which Foundation counts as zero days. Swift then stays one quote behind every day until the app republishes. In publication s01648 (Santiago, change at 00:59, last published instant 6 September 2026) the lag was still present at the last query, nine days later. Kotlin builds its civil-day number from year, month and day, so it does not lag. Two of the seven zones, São Paulo and Tehran, contribute only through the historical transitions.

D5. Timeline entry dates and the step rule disagree (Swift display; 3 zones, 9 publications). Past the published instants, Swift's timeline places entries with `nextDate(…, matchingPolicy: .nextTime)` but evaluates each entry with `dailyStep`, which uses `bySettingHour`. When a change time is skipped and the two calls resolve it to different instants, an entry can be evaluated just before the step it was meant to show. The widget then displays the previous quote until the next entry. In Nuuk, with a change set for 23:00, clocks jumped from 23:00 on 28 March 2026 straight to 00:00 on 29 March, and the displayed quote stayed one behind the rule from 00:00 to 22:59 on 29 March.

D6. The next alarm keeps a shifted wall-clock time (Kotlin display; 21 zones, 108 publications). After a gap day, Kotlin's `nextChange` takes the boundary it has just used, for example 03:30 instead of the configured 02:30, and calls `Calendar.add(DATE, 1)`. That keeps the shifted time, so the next re-render is late on the following day by the length of the gap: an hour in most zones, 30 minutes at Lord Howe. The rule itself is right, and the widget shows the correct quote once the late alarm fires. Like D3, this was observed on a desktop JVM and needs confirming on Android.

D7. Dart's elapsed count disagrees with its own published list (Dart app; 2 zones, 17 publications). In Nuuk, a change set for 23:30 on the night clocks jump from 23:00 to 00:00 is moved by `DateTime` to 00:30 the next day, and the published list carries 00:30. `elapsedDailyChanges` counts by calendar date and treats that day's change as already passed at 00:00. For 30 minutes, the app is one quote ahead of both widgets and of the list it published itself. The second zone, Apia, contributes only through Samoa's historical 2011 change.

One further class falls outside the eight. In 5 publications, Swift's day count also lagged across Samoa's skipped 30 December 2011. That is historical and can no longer happen.

Table 5 summarises the eight. The likelihood column is a judgement from the trigger conditions; I have no usage data with which to measure it. The last column counts publications whose transition is in 2025 or later, to separate current rules from the historical stress cases.

Table 5. User-visible impact and likelihood of each divergence.

ID	What the user sees	How long	Needs	Likelihood	Publications (current rules)
D1	Nothing visible	under 1 s	any interval widget	every period, but invisible	1,184 (1,125)
D2	Widget steps back a quote when the app opens	usually 1 h; can last the whole publication	Today override, clock change within a day of the first change	rare	58 (47)
D3	Android widget changes an hour late	1 h, once	tail, change time in the repeated hour	rare; unconfirmed on Android	68 (58)
D4a	iPhone widget changes early	up to the gap, once	tail, change time in the skipped hour	rare; unconfirmed on iOS	50 (48)
D4b	iPhone widget one quote behind	every day until the app is opened	tail ending on a midnight clock-change day	rare, but long-lasting; unconfirmed on iOS	61 (47)
D5	iPhone widget shows the previous quote	most of one day	tail, skipped change time in Nuuk, Troll or Lord Howe	very rare; unconfirmed on iOS	9 (9)
D6	Android widget updates late	up to 1 h, the day after a gap	tail, change time in the skipped hour	rare; unconfirmed on Android	108 (93)
D7	App one quote ahead of its widgets	up to the gap, once a year	change time in a late-evening gap (Nuuk)	very rare	17 (8)

5.3 The documented rule is not the shipped rule

`currentPoolIndex`, the function both native files claim to mirror, agreed with the app on only 89.29% of queries. For intervals it disagrees exactly where D1 does, because it

receives the truncated anchor. For daily clock times it uses a fixed 24-hour interval, so it drifts by an hour across every clock change (4,488 publications). It is not called anywhere in `lib/`. The native comments therefore point to a function that no longer describes the app's behaviour, and one of the contract tests checks that the Kotlin file contains its name.

5.4 Latent divergences in payloads the app does not publish

I also gave both native harnesses nine hand-built payloads that the current Dart publisher never produces, such as legacy or malformed data. Seven made Swift and Kotlin disagree (Table 6). Two legacy tier-filtering cases agreed at all three widget tiers. None of these is a live bug. Each is a place where the two copies would split if the publisher changed.

Table 6. Hand-built payloads (UTC). Disagreeing answers out of queries.

Payload	Swift	Kotlin	Disagree
Unsorted change instants	`prefix` stops at first later instant	`count` scans all	6/13
Anchor epoch 0	rotates from 1970	treats as unset, index 0	6/8
Fractional interval (90.5 s)	Double	truncated to 90 s	19/40
Fractional change instants	Double	`getLong` truncates	12/20
IDs `café` (NFC, one code point) and `café` (NFD, two)	equal under canonical equivalence	different code units	2/6
Interval stored as a string	ignored, manual	parsed	4/6
Anchor keys missing	falls back to ISO `updated_at`	index 0	4/6

The Unicode case deserves a note. Swift's `String ==` uses canonical equivalence, so two IDs that differ only in normalisation compare equal. Swift's manual "Change quote" intent finds the current quote with `firstIndex`, so it would treat two such quotes as one. All 12,759 IDs in the bundled library are ASCII, so this cannot happen today. Quote length tiers are computed once in Dart and shipped as integers, which removes another possible Unicode divergence: UTF-16 length against grapheme count.

5.5 String-matching contracts against the differential test (RQ3)

All 22 contract tests passed on the shipping code that contains D1 to D7. Six of the 22 touch the pool or its rotation. None refers to `scheduleDates`, `dailyStep`, `scheduledIndex`, `dailyBoundary` or `nextChange`, the functions where the native divergences live. So the answer to "would they have caught it?" is no for every divergence in Section 5.2.

The mutant study makes the comparison concrete (Figure 4). The contract tests detected 0 of 12 behaviour-changing mutants. They did flag both behaviour-preserving refactors: renaming `currentPoolIndex` broke one test, and rewriting `$0.tier <= maxTier` as `maxTier >= $0.tier` broke another, although nothing a user sees changed. The differential test detected 12 of 12 behaviour-changing mutants, each changing between 168 and 95,419 of the 220,228 answers, and flagged neither refactor.

Twelve mutants is a small number. If they were a random sample of behaviour changes, the 95% Clopper–Pearson interval for the contract tests' detection rate would be 0% to 26.5%, and for the differential test 73.5% to 100%. They are not a random sample: I wrote them by hand, knowing the code, so these intervals indicate only how little twelve cases can establish. Together with the eight real divergences, the string-matching tests caught 0 of 20 behaviour differences in this study.

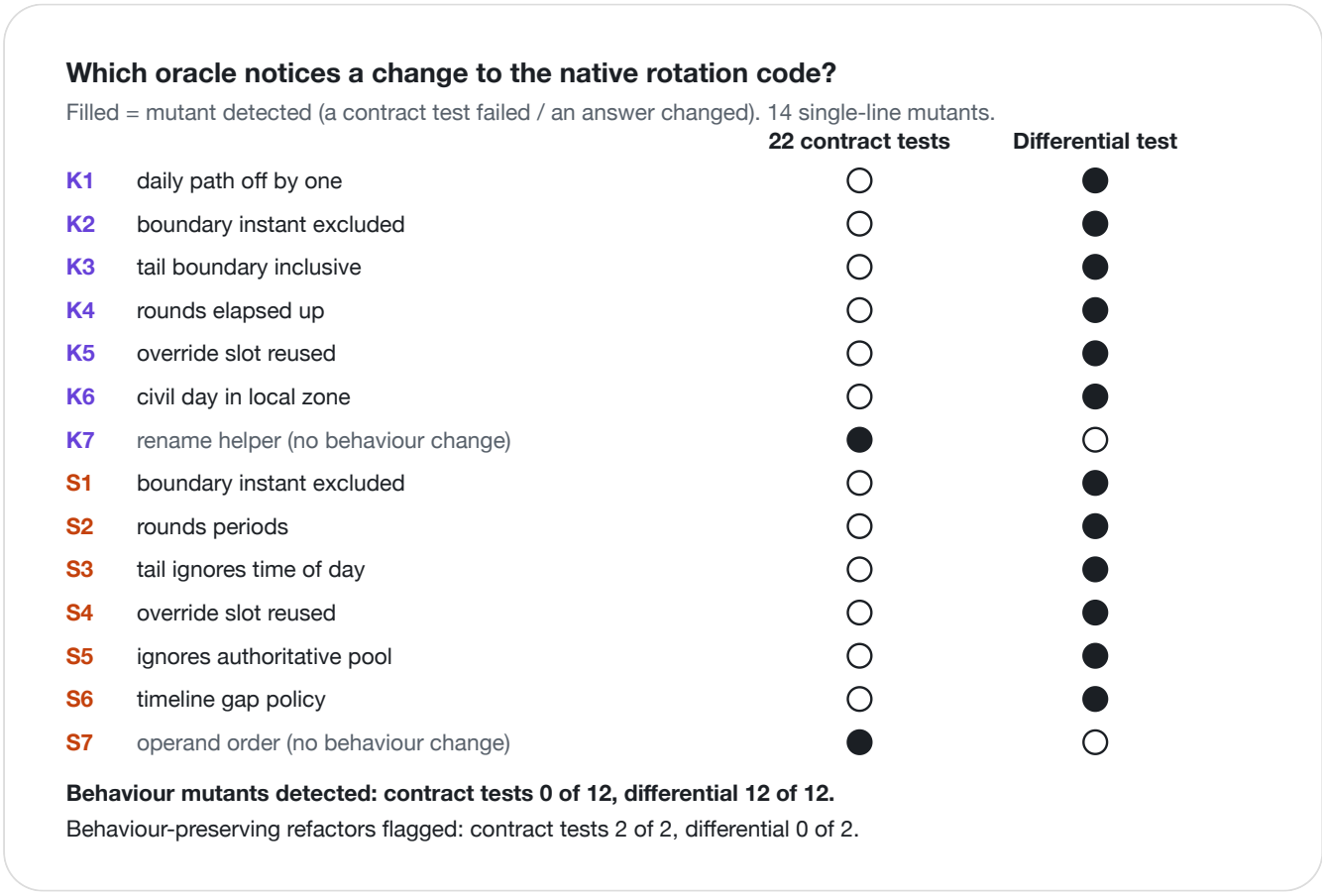


Figure 4. Mutant detection by oracle. Filled circles are detections.

5.6 Cost (RQ4)

The full run took 6.5 s of CPU in Dart, 91.0 s in Swift and 195.3 s on the JVM, summed over 29 processes per runtime. That includes process start-up and, for the native harnesses, simulating every timeline and alarm chain. In total it is about 0.53 ms of CPU per query instant for all six answers. These figures come from a single run on one machine that was heavily shared, with load averages between 68 and 112 throughout, so they indicate scale rather than a precise cost, and wall-clock time is not reported. At that rate a trimmed set of a few thousand queries in six zones would need a few seconds of CPU plus process start-up, which is cheap enough for continuous integration.

---

6 Discussion

Contracts that check spelling, and tests that check behaviour

The 22 contract tests were a reasonable answer to a real constraint: the Dart test runner cannot execute Swift or Kotlin. They guard the particular regressions they were written for, such as a removed accessibility label or a reintroduced shadow layer. For layout they are probably the right tool.

For calendar logic they are the wrong tool. A string test can confirm that a line is present, but not what the line computes. That limit cuts both ways. All 12 behaviour-changing mutants passed, and both harmless refactors failed, because the tests pin exact spellings. One of them pins the name of a function the app no longer uses. A string test gives assurance about the code's shape, not its behaviour.

An executable differential test removes the constraint instead of working around it. Swift and Kotlin both compile and run on a development Mac. The rotation functions are pure apart from `Calendar.current` and `System.currentTimeMillis()`, so they can run outside the app. The effort went into extraction, not into the testing idea.

Why the copies diverged

The Dart publisher already does the right thing for the first 96 changes. It computes change instants in Dart and ships them as absolute times. In that region the native copies only count, and they almost always agree. Every native divergence appears where a copy computes local time itself: past the end of the list, or when choosing when to wake up. Each calendar library answers "02:30 on a day without 02:30" and "01:30 on a day with two" differently, and each answer is documented, defensible library behaviour. None of these is a bug in Dart, Foundation or Java. The bug is assuming that their answers match.

The two Dart-side divergences have the same shape. D2 uses a 24-hour `Duration` where the rest of the code uses calendar days. D7 counts calendar dates where the list it published counts instants.

Practical implications

1. **Publish further ahead instead of computing past the list.** Let one runtime do all calendar arithmetic and ship absolute instants. Adelante already does this for 96 changes. Publishing a year or more ahead costs little: a year of daily instants as epoch seconds is about 4 KB of JSON. When the list does run out, the widget can show the last quote and ask the user to open the app, rather than extrapolate. This alone would remove D3 to D6.
2. **Use one notion of a day.** Within one codebase, choose civil-calendar arithmetic or fixed durations and keep to it. D2 is the cost of mixing them.
3. **If the native side must extrapolate, specify the policy.** State the gap and overlap rule explicitly (for example, "first occurrence; skipped times move forward by the gap") and implement it, rather than relying on each library's default.
4. **Test the copies against each other, not against their text.** Keep string contracts for layout. For logic, run a small differential harness in continuous integration on a trimmed set of boundary cases.
5. **Keep comments that claim a mirror honest.** If the mirrored function is not on the production path, either route production through it or delete it.

For the app

I did not change Adelante. For the owner, in priority order:

- D4b: a one-quote lag that lasts until the app is opened, in midnight-change zones when the three-month schedule runs out on a change day;
- D5: a stale display for most of a day; D6: a late update after a gap day;
- D2: a visible step back for overridden widgets around clock changes;
- D3, D4a and D7: windows of 30 to 60 minutes;
- D1: sub-second and invisible.

The smallest fix that removes most of them is to publish further ahead and make the native tail count published instants only. That is better than computing local time in two more languages. The Kotlin findings, D3 and D6, should be confirmed on an Android device before they are fixed.

7 Threats to validity

Harness fidelity (internal). The harnesses run copies, not the shipped binaries. The copied ranges are marked and the Swift ranges were checked line by line, but the glue around them is mine. I could have changed how a value is decoded, for example how the catalog JSON reaches `doublePref`. I mirrored the production decoding paths (`JSONSerialization` on iOS, the catalog-backed `SharedPreferences` override on Android) and probed each divergence's library call on its own. The Dart app answer is extracted arithmetic, not the Riverpod provider. The app also rebuilds its queue when the UTC offset changes; I reasoned from the source, but did not test, that this rebuild keeps the current quote.

Classification (internal). The root-cause classes are assigned by answer pattern and calendar facts, not proven query by query. A different cause with the same pattern would be counted in the same class. I confirmed one example per class with a library probe and inspected several more by hand, but not all of them.

Runtime differences (construct). Kotlin ran on a desktop JVM, not on Android's runtime. Android's `java.util.Calendar` exposes the same API, but its implementation and time-zone data differ, so D3 and D6 may not occur, or may occur differently, on a phone. Swift ran on macOS. Foundation on macOS belongs to the same framework family as on iOS but is not the same build, so D4a, D4b and D5 also need confirming on an iPhone or simulator. The JVM's tzdb (2026a) is older than the macOS data (2026e). No divergence I found depends on a rule that changed between those versions, but I did not check every zone.

Oracle choice (construct). Treating the Dart app as the reference makes "divergence" mean "the widget will jump when the app opens". In D2 and D7 the two native copies are arguably closer to what the user intended than the app is. The classification reports who deviates, not who is right.

Idealised display timing (construct). Real WidgetKit reloads and Android alarms are later and less regular than I assumed, so displayed-quote staleness is a lower bound.

Input bias (external). The 7,658 publications and 29 zones are chosen inputs, not real usage. Queries cluster at boundaries by design, and some transitions are historical. Agreement rates describe hard cases. They are not the probability that a given user sees a wrong quote on a given day, which is far smaller.

Timing (conclusion). The CPU times come from one run on one heavily loaded machine. They show the order of magnitude, not a stable measurement.

8 Limitations

This is one rule in one app by one developer. Seven of the eight root causes are calendar arithmetic, because that is what this rule does; logic involving hashing, sorting or floating point would need different inputs. I did not test the Watch app, which uses a separate Swift-only reading deck with its own SplitMix64 generator, the lock-screen surface offset, or the per-period manual selection.

No phones, simulators or emulators were used, so every native finding is a finding about the copied code on desktop runtimes. The mutant set is small and hand-written by the person who knew the code, so the detection counts illustrate the difference between the oracles rather than measure it. The likelihood judgements in Table 5 rest on trigger conditions, not usage data. The study did not fix anything; whether each divergence is worth fixing is a product decision for the owner.

9 Conclusion

Three copies of one small rule, each written to mirror the others, agreed on 95.66% of deliberately hard test instants. Where they disagreed, the reason was usually the same: a copy computed local time itself, and its language's calendar library made a different choice about skipped and repeated hours. The project's 22 string-matching contract tests passed throughout, and they also passed for all 12 hand-written behaviour-changing mutants. A differential harness of a few hundred lines per language detected every one. These results come from one app and from desktop runtimes, and the Kotlin findings in particular still need an Android device. Within those limits, the lesson for shared logic across native targets is plain: compute instants in one place and publish them far enough ahead, and test what the copies compute, not how they are written.

Artifact

The artifact at /research/one-rule-three-runtimes/artifact/ contains:

- the input generator and the latent-payload generator;
- the Dart, Swift and Kotlin harnesses, with the copied ranges marked and SHA-256 hashes of the source files;
- the run, analysis, mutant and figure scripts;
- the generated inputs (`scenarios.json`);
- the per-zone raw answers of all three runtimes;
- `summary.json`, `mutants.json` and `cost.json`;
- a README with commands and toolchain versions.

Adelante's source is private and is not included. The copied rotation functions are the only app code in the artifact. Rerunning the contract tests and mutants needs an Adelante checkout at commit `af1aea2`.

References

1. W. M. McKeeman. Differential testing for software. Digital Technical Journal 10(1), pp. 100–107, 1998.
2. A. Avizienis. The N-version approach to fault-tolerant software. IEEE Transactions on Software Engineering SE-11(12), pp. 1491–1501, 1985. doi.org/10.1109/TSE.1985.231893
3. J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. IEEE Transactions on Software Engineering SE-12(1), pp. 96–109, 1986. doi.org/10.1109/TSE.1986.6312924
4. S. Tiwari, S. Chen, A. Joukov, P. Vandervelde, A. Li and R. Padhye. It's about time: an empirical study of date and time bugs in open-source Python software. In 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), pp. 39–51, 2025. doi.org/10.1109/MSR66628.2025.00020
5. X. Yang, Y. Chen, E. Eide and J. Regehr. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11), pp. 283–294, 2011. doi.org/10.1145/1993498.1993532

6. T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse and Z. Q. Zhou. Metamorphic testing: a review of challenges and opportunities. ACM Computing Surveys 51(1), article 4, 2018. doi.org/10.1145/3143561

7. S. Segura, G. Fraser, A. B. Sanchez and A. Ruiz-Cortés. A survey on metamorphic testing. IEEE Transactions on Software Engineering 42(9), pp. 805–824, 2016. doi.org/10.1109/TSE.2016.2532875

8. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), pp. 268–279, 2000. doi.org/10.1145/351240.351266

9. E. Lear and P. Eggert. Procedures for maintaining the time zone database. RFC 6557 (BCP 175), IETF, 2012. doi.org/10.17487/RFC6557

10. M. Fazzini and A. Orso. Automated cross-platform inconsistency detection for mobile apps. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 308–318, 2017. doi.org/10.1109/ASE.2017.8115644

11. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), pp. 649–678, 2011. doi.org/10.1109/TSE.2010.62

12. R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser. Are mutants a valid substitute for real faults in software testing? In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp. 654–665, 2014. doi.org/10.1145/2635868.2635929

Cite and reuse

Authorship and AI assistance. Yousof Selim designed and built Adelante and directed this study; much of the implementation and parts of this report were produced with AI coding assistants (Claude, Codex) under his direction and review; experiments are reproducible from the artifact.

```
@techreport{selim2026one,
  author      = {Yousof Selim},
  title       = {One Rule, Three Runtimes: Differential Testing of a Dai
  institution = {Sunway University (Lancaster University dual degree), M
  type        = {Technical report},
  number      = {v1.0},
  year        = {2026},
  url         = {https://yousofselim.com/research/one-rule-three-runtime
}
```