



(RESEARCH ARTICLE)



SYSTEMATIZING APPROACHES TO TESTING ASYNCHRONOUS IOS APPLICATION CODE FROM DETERMINISTIC TEST DOUBLES TO COMPILE-TIME CHECKING

Tomasz Kubiak *

Welltech, Krakow, Poland.

* Corresponding Author

ORCID Details

Tomasz Kubiak: <https://orcid.org/0009-0006-1245-8838>

International Journal of Science and Research Archive, 2026, 21(01), 477–487

Publication history: Received on 02 September 2026; revised on 12 October 2026; accepted on 14 October 2026

Article DOI: <https://doi.org/10.30574/ijrsra.2026.21.1.1803>

Copyright © 2026 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0

ABSTRACT

Asynchronous execution is the default condition of iOS application code, and tests of such code inherit an ordering that the source does not fix. A test may pass on a workstation and fail on a continuous integration runner, and a defect that appears once in fifty runs is hard to separate from a defect that appears once in fifty user sessions. Two lines of tooling address this condition from opposite ends. Swift moved a class of concurrency checks into the compiler through Sendable conformance and actor isolation, while the test ecosystem developed injectable clocks and schedulers that advance virtual time by explicit steps. This structured analytical review examines what each mode verifies and what neither reaches. Four recurring failure scenarios drawn from first-hand production experience are decomposed into five dimensions that a test must control, namely time, scheduling, dependencies, cancellation, and shared mutable state. Nine verification approaches are then coded against these dimensions and against the phase at which each check runs. The resulting coverage matrix shows that compile-time checking guarantees freedom from memory-level data races and expresses nothing about ordering, staleness, or cancellation, whereas deterministic tests control those properties only along the interleavings a test exercises. The gap between the modes closes through design for injection at module boundaries and through step-wise assertions on state, and the review derives the structure of such a test environment. The practice of lengthening waits is treated as an observed industry pattern whose cost is delay without control.

Keywords: Actor Isolation, Asynchronous Testing, Flaky Tests, Sendable, Swift Concurrency, Test Schedulers.

1 INTRODUCTION

Asynchronous execution has become the ordinary condition of iOS application code. Network requests, sensor and health data streams, persistence, and navigation transitions complete at moments the caller does not control, and user interface state is assembled from results that arrive in an order the source code leaves open. Tests of such code inherit this openness. A test that passes on a developer workstation may fail on a continuous integration runner, and a test that fails once in fifty runs is difficult to separate from a defect that appears once in fifty user sessions. The software engineering literature classifies such tests as flaky, and asynchronous waiting has remained among the leading root

causes in the surveys published since the first large empirical analysis of the phenomenon [1,2]. Research on automated repair reports that developers respond to an asynchronous flaky test by inserting or lengthening a wait, and that such repairs stay unreliable because the wait encodes a guess about machine speed [3]. A study of fifty-two projects under twenty-seven resource configurations found that 46.5% of flaky tests were sensitive to the CPU and memory available at run time, which explains why a suite that is green on a laptop turns red on a shared runner [4].

Two developments make the iOS case worth a dedicated review. First, the Swift language moved a class of concurrency checks from run time to compile time between 2021 and 2024. Asynchronous functions, structured tasks, actors, and the Sendable protocol entered the language in Swift 5.5 [5–7], and the Swift 6 language mode made strict concurrency checking the default, so that a mutation of shared state across an isolation boundary became a build error [8]. Second, the test-side ecosystem produced injectable clocks and schedulers that let a test advance virtual time by explicit steps and assert on state between them. These two lines of work approach the same problem of untrusted asynchronous code from opposite ends, and practitioner discussion tends to treat them as substitutes.

This review argues that the two modes verify different properties and that a gap remains between them. The argument rests on first-hand observation of production iOS code in which concurrency defects took four recurring shapes. A slower request issued first completed last and overwrote newer state. Competing requests produced an interface that matched neither of them. Data was read before the concurrent work that produced it had finished. A request that was never cancelled completed after the user had moved on and triggered a screen that was no longer relevant. Direct experience with test suites for such code adds a fifth observation, this time about practice. Combine-based tests were made to pass with sleeps of one to five seconds, and a single group of simple unit tests added about a minute to the suite. These figures record observation and carry no claim of measurement. The review treats them as motivation for the analysis and as one illustration of a pattern the literature documents across ecosystems, in which teams wait longer where they could take control of time, scheduling, dependencies, and cancellation.

Two earlier contributions from the same line of work addressed adjacent questions. A reproducible protocol for classifying UIKit-to-SwiftUI migration treated changes to tests as one attribute of a migration episode without asking what those tests were able to verify [9]. A local-first architecture for personalization in health and fitness applications moved signal processing onto the device and thereby increased the share of asynchronous work whose ordering matters for correctness [10]. Neither examined the verification question that the present review takes up, and neither built its argument on an axis between compile-time and run-time checking.

The research question is threefold. Which properties of asynchronous iOS code does each verification mode guarantee, which properties does neither mode reach, and what structure of the test environment follows from that division? To the extent that the accessible literature shows, no prior review has mapped Swift verification approaches onto the dimensions that a test must control. The contribution is, first, a catalogue of four failure scenarios decomposed into the dimension each left uncontrolled, second, a coverage matrix that places nine approaches against five dimensions and three phases, third, an explicit description of the blind spot of each mode, and fourth, a structure for a deterministic test environment derived from the matrix.

2 MATERIALS AND METHODS

2.1 Study Type and Evidence Architecture

The study is a structured analytical review. It does not claim the completeness of a systematic literature review, and it reports no counts of primary studies. Its purpose is to make an argument falsifiable by decomposing observed failures into named dimensions and by coding public verification approaches against those dimensions, so that a reader can dispute a cell of the resulting matrix with reference to a documented mechanism.

The evidence base combines four corpora, and Table 1 records the contents and the analytical function of each. The corpora cover research on test flakiness and its repair [1–4,11–13], language-design research and a practitioner reference on Swift concurrency [14–16], Swift Evolution proposals and official Apple documents [5–8,17–20], and open-source testing libraries and framework documentation [21–26].

Table 1: Evidence corpora and their analytical function

Corpus	Contents	Analytical function
Flakiness research	Survey and multivocal review of flaky tests, automated repair of async flaky tests, resource sensitivity, developer experience, UI flakiness, Android event-order exploration	Establishes asynchronous waiting as a documented root cause and characterizes the waiting practice and its repair
Language-design research	Type system for fearless concurrency, empirical study of Kotlin coroutine defects, practitioner reference on Swift concurrency	Supplies the semantics of compile-time isolation and a cross-ecosystem comparison
Language and platform documents	Swift Evolution proposals SE-0296, SE-0302, SE-0304, SE-0306, SE-0337, SE-0414, the Swift 6 migration guide, WWDC session on data races	Defines what the compile-time mode guarantees and what it leaves to the programmer
Open-source testing tools	Test clocks, Combine test schedulers, step-wise store tests, XCTest expectations, Swift Testing, thread sanitizer	Defines what the run-time mode controls and at which phase

No corporate data, product metric, or internal code enters any corpus. The four failure scenarios and the two figures about waiting come from first-hand experience, are described at the level of mechanism, and are attributed to no product, employer, or codebase.

2.2 Analytical Procedure

The unit of analysis is a property of an asynchronous code path that a verification technique may or may not guarantee. The procedure has three steps. In the first step each failure scenario is decomposed into a mechanism, an observable symptom, the dimension whose lack of control allowed the mechanism to operate, and a minimal construction that reproduces the failure in a deterministic test. Five dimensions were derived from the scenarios and from the flakiness literature. Time denotes the passage of wall-clock or virtual time on which delays, debounces, and timeouts depend. Scheduling denotes the order in which queued work is executed and on which executor. Dependencies denote the collaborators whose behavior a component consumes and whose replacement a test requires. Cancellation denotes the propagation of a stop signal to in-flight work and the observability of that propagation. Shared mutable state denotes memory reachable from more than one isolation domain at once.

The four scenarios were selected from first-hand experience under three criteria. Each scenario had produced a defect visible to a user, so that the review does not argue from hypothetical interleavings. Each scenario had been observed in more than one codebase and in more than one concurrency model, since the stale overwrite and the missing cancellation appeared in callback-based code, in Combine pipelines, and in async functions alike. Each scenario had a counterpart in the published literature, so that its inclusion does not rest on one practitioner's memory.

In the second step each verification approach is coded against the five dimensions. A cell receives the value full when the approach makes the dimension deterministic for any code path that compiles or runs under it, partial when the approach controls the dimension only along the paths that a test exercises or only under a build configuration that production does not use, and none when the approach does not address the dimension. Each approach is also assigned a phase, which is the moment at which its check produces a verdict. The three phases are compile time, the deterministic test run, and the production run. Coding relied on the documentation of each approach and on the semantics stated in the relevant language proposal.

In the third step the coded matrix is read along both axes. Reading a column identifies which approaches reach a dimension at all, and reading a row identifies which dimensions an approach cannot reach. Cells that no approach fills at a given phase define the gap that the review sets out to describe. The structure of a deterministic test environment is then derived from the union of the cells that the run-time approaches fill, with each injected seam corresponding to one dimension.

3 RESULTS

3.1 Two Lines of Tooling on One Axis

The compile-time mode and the run-time mode developed in parallel over the same four years, and the sequence in which they arrived explains part of the waiting habit that the review examines. Fig. 1 places the milestones of both lines on one axis. On the compiler side, Swift 5.5 introduced asynchronous functions with the `async` and `await` keywords [5], structured concurrency with task groups and child tasks whose lifetime is bounded by a scope [17], actors as reference types whose mutable state is reachable only through serialized asynchronous access [6], and the `Sendable` protocol as a marker that a value may cross an isolation boundary without a data race [7]. Swift 5.6 introduced an incremental migration path with a warning flag and a `preconcurrency` attribute for modules that had not adopted the model [18], and Swift 5.7 exposed the same checking as three levels, minimal, targeted, and complete, into which a module opts, so that warnings could accumulate without breaking the build. Swift 5.10 completed strict checking under a flag, and the Swift 6 language mode made the complete level the default and turned the remaining diagnostics into errors [8].

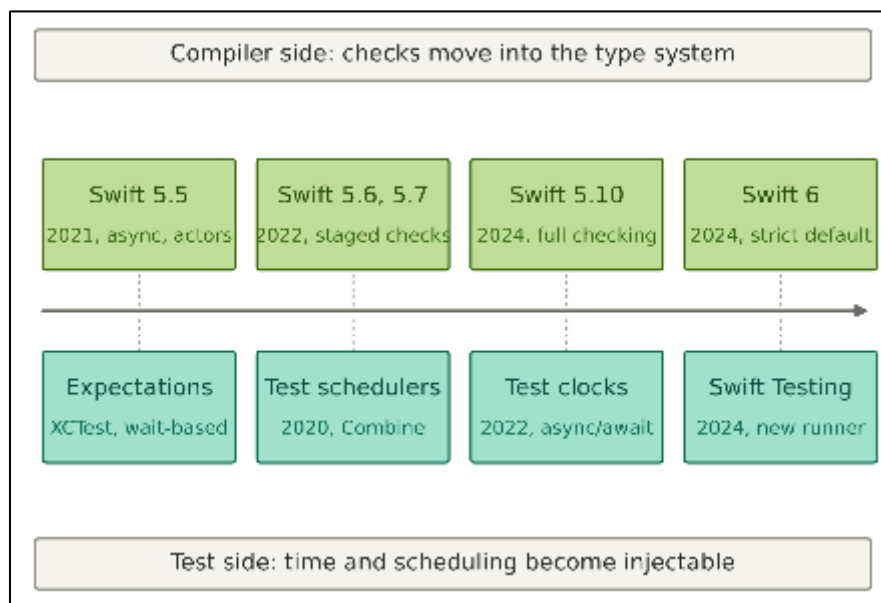


Figure 1: Milestones of compile-time concurrency checking and of test-side time control

On the test side, the oldest instrument is the expectation object of XCTest, which blocks the test until a condition is fulfilled or a timeout elapses [24]. The expectation is a wait, and its timeout is the guess about machine speed that the repair literature identifies as the root of asynchronous flakiness [3]. Test schedulers for Combine appeared in 2020 as an open-source library that replaces the dispatch queue behind a publisher with a scheduler whose time advances only when the test calls an advance method [22]. Clocks for the `async` and `await` model followed in 2022, when a library defined a clock protocol conforming implementation whose sleep calls resolve only when the test advances virtual time [21]. Swift Testing arrived in 2024 as a new runner with native support for asynchronous test functions and for parallel execution [25]. The lag between the two lines is visible in Fig. 1. Codebases that adopted the new model in its first year had a full year in which the only instrument that reached code written against `async` functions was the expectation with a timeout, since Combine test schedulers applied to publishers alone, and the waits written in that year did not disappear when the clock arrived.

The thread sanitizer sits between the two lines. It instruments memory accesses at run time and reports two unsynchronized accesses to the same location when at least one is a write [26]. Its verdict depends on which interleavings the run happens to produce, and the instrumented build is not the build that ships. The sanitizer therefore belongs to the run-time mode by phase and to the compile-time mode by the class of property it targets, and the coding in the next subsections reflects that position.

3.2 Four Failure Scenarios and the Dimension Each Left Uncontrolled

The four scenarios that motivate the review are decomposed in Table 2 into mechanism, symptom, uncontrolled dimension, and a minimal deterministic reproduction. Each row records what a test would have needed to control in order to observe the failure on every run. The scenarios recur in the literature under other names, and the table cites the closest documented counterpart where one exists.

Table 2: Failure scenarios decomposed into mechanism, symptom, uncontrolled dimension, and deterministic reproduction

Scenario	Mechanism and symptom	Uncontrolled dimension	Deterministic reproduction
Stale response overwrites newer state	Handler treats completion order as issue order, interface shows results for an outdated query	Scheduling	Inject clock or scheduler, issue two requests, advance so the second completes first, assert the first completion is discarded
Competing requests produce an inconsistent interface	Two sources write partial updates to overlapping state, screen mixes fields from two versions	Scheduling, shared mutable state	Drive both sources through injected schedulers, assert composed state equals one consistent version after every advance
Read before concurrent work completes	Missing happens-before between producer and consumer, stale display or crash on partial write	Scheduling, shared mutable state	Schedule refresh, advance to before completion and assert old value, advance past completion and assert new value
Missing cancellation triggers stale navigation	Request outlives the screen and its handler owns navigation, unexpected screen transition	Cancellation	Issue request through injected clock, trigger disappearance, advance past completion, assert no navigation action was emitted

The first scenario is a stale response that overwrites newer state. A view model issues a request for a query, the user refines the query, and a second request is issued before the first returns. When the first request completes last, its handler writes the result of the outdated query over the result of the current one. A deterministic reproduction injects a scheduler or clock, issues the two requests, advances time so that the second completes first, and asserts that the first completion is discarded. Combine test schedulers and the async clock both support this construction, and the composable architecture test harness makes the discarded effect visible as an unreceived action [21–23].

The second scenario is a pair of competing requests that produce an interface matching neither. Two independent sources update overlapping regions of state, such as a remote profile and a local cache, and each writes its own view of the region without regard to the other. Under strict concurrency checking, the shared-state half of this scenario is a compile error if the region is a class instance passed across isolation domains, and the scheduling half remains a test concern.

The third scenario is a read that occurs before the concurrent work producing the value has finished. A screen reads a cached value on appearance, a background task is refreshing that value, and the read observes either the old value or a partially written one. The compile-time mode addresses this scenario to a greater extent than the others, because an actor-isolated cache makes a partially written value unreachable, and a read that awaits the actor observes either the old value or the new one. The ordering between old and new remains a run-time property, and the deterministic reproduction schedules the refresh, advances to the point before its completion, reads, and asserts on the old value, then advances past completion and asserts on the new one.

The fourth scenario is a request that is never cancelled and completes after navigation. A detail screen issues a request, the user returns to the list before the request completes, and the completion handler pushes a screen that presents the detail for an item the user has left. Structured concurrency provides the propagation mechanism, since a child task is cancelled when its parent scope ends [17], and the compiler enforces nothing about whether a handler checks for cancellation before acting. The composable architecture harness fails a test in which an effect is still in flight at the end of the test, which converts a missing cancellation into a deterministic failure [23].

Two documented counterparts confirm that the four scenarios are not an artifact of one codebase. The empirical analysis of two hundred thirty-five flaky user interface tests across web and Android projects reports asynchronous waiting and animation timing among the root causes, and it records increased delays between actions as one of the common fixing strategies, which is the waiting habit that Table 2 sets against deterministic reproduction [12]. A study of fifty-five concurrency defects in Kotlin coroutine programs, including a mobile browser and an e-commerce application, lists mishandled cancellation and work that outlives its scope among the patterns tied to coroutine semantics, which correspond to the fourth scenario and to the first in another language [15].

3.3 Coverage of Controlled Dimensions by Verification Approach

Table 3 codes nine approaches against the five dimensions, and the rows are grouped by the mode to which each approach belongs. The first group contains the waiting practice, the second contains deterministic run-time doubles, the third contains run-time instrumentation, and the fourth contains compile-time checking. The value in each cell follows the coding rule stated in the method.

Table 3: Coverage of controlled dimensions by verification approach

Approach	Phase	Time	Scheduling	Dependencies	Cancellation	Shared state
<i>Waiting without control</i>						
Sleep or expectation with timeout	Test run	none	none	none	none	none
<i>Deterministic run-time doubles</i>						
Mock dispatch queue	Test run	partial	full	none	none	none
Test scheduler for Combine	Test run	full	full	none	partial	none
Test clock for async and await	Test run	full	partial	none	partial	none
Step-wise store test	Test run	full	full	full	full	partial
Dependency injection at module boundary	Test run	partial	partial	full	none	none
<i>Run-time instrumentation</i>						
Thread sanitizer	Test run, instrumented	none	none	none	none	partial
<i>Compile-time checking</i>						
Sendable and actor isolation	Compile	none	none	none	none	full
Swift 6 language mode	Compile	none	none	partial	none	full

Waiting, whether as a sleep or as an expectation with a timeout, controls nothing. It lets real time pass and hopes that the work completes within the allowance. A sleep of one second on a fast machine and a sleep of one second on a loaded runner differ in the amount of work that completes, and the study of computational resources quantifies the consequence, since almost half of the flaky tests in its corpus changed failure rate with CPU and memory [4]. The cost of the practice is delay without control.

A mock dispatch queue replaces the queue on which a component enqueues work with an object that stores closures and executes them when the test calls a run method. It makes scheduling deterministic within the component that uses it and leaves time uncontrolled, because a closure scheduled after a delay still depends on a real timer unless the mock also intercepts the delay. A test scheduler for Combine subsumes the mock queue and adds control of time, since a publisher that debounces or delays does so against the scheduler that the test advances [22]. A test clock for the async model controls time in full and scheduling in part, since sleep calls resolve on advance while the order of resumption among tasks that become ready at the same instant remains a property of the executor [21].

The step-wise store test is the only run-time approach that receives full coding on four dimensions. It controls time and scheduling through the clock and scheduler it injects, controls dependencies through an injected dependency container, and controls cancellation because an effect still in flight at the end of a test is a failure [23]. Dependency injection at the module boundary receives full coding for dependencies and partial coding for time and scheduling, because an injected clock and scheduler are dependencies in the sense that matters here, and a boundary that admits them admits control of the two dimensions. First-hand experience with dependency graphs that admitted no replacement at the boundary is what places this approach in the table. When a collaborator could not be substituted at the level where the test

operated, the test had to alter behavior several layers deeper in the graph, and the resulting construction was fragile and slow to write.

Sendable conformance together with actor isolation receives full coding for shared mutable state. A type that crosses an isolation boundary must be Sendable, an actor serializes access to its own state, and a violation is a diagnostic at compile time for every code path in the module, exercised or not [6,7]. The same approach receives none for time, scheduling, dependencies, and cancellation, because the type system expresses no ordering between two asynchronous completions, no notion of virtual time, and no obligation to check for cancellation. The Swift 6 language mode receives full coding for shared mutable state and partial coding for dependencies, since strict checking forces a codebase to declare which collaborators may cross boundaries and thereby documents the seams that a test could replace, without providing the replacement [8].

Fig. 2 reads the same evidence along the other axis. It places the three phases across the top, groups the five dimensions into four property classes down the side, and marks each cell with the approach that reaches it. Ordering and staleness, cancellation, and timing are reached by deterministic tests and by nothing else. The cells in the compile-time column for those three classes are empty because the type system cannot express the properties, and every cell in the production column is empty because the properties reach the user as a symptom before any check observes them.

	Compile time	Deterministic test	Production run
Data races memory level	Sendable and actor isolation	Exercised paths, ThreadSanitizer	Seen as crashes or corrupted state
Ordering state overwrites	Not expressible in the type system	Step-wise state assertions	Seen as wrong UI by the user
Cancellation late completion	Not expressible in the type system	Observable cancellation	Seen as stale navigation
Timing delays, debounce	Not expressible in the type system	Virtual clock and scheduler	Seen as flaky tests on slower CI

■ Guaranteed by the compiler
 ■ Controlled by the test
 Gap: reaches the user as a symptom

Figure 2: Verification phases against property classes of asynchronous code, with dashed cells marking properties that reach the user as a symptom before any check observes them

3.4 The Gap Between the Modes

Reading Table 3 and Fig. 2 together isolates two blind spots, one per mode. The compile-time mode guarantees the absence of memory-level data races and expresses nothing about the order in which two asynchronous operations complete. Actor methods are reentrant, which means that when an actor-isolated function suspends at an await, another call on the same actor may run before the first resumes, and the state that the first call observed before the suspension may have changed by the time it continues [6]. A view model implemented as a main-actor class compiles under Swift 6 and still overwrites newer state with an older response, because both writes are serialized on the same actor and the serialization says nothing about which write is correct.

Cancellation is in the same position. Structured concurrency propagates a cancellation flag to child tasks, and the compiler enforces that the flag exists [17]. Whether a handler checks the flag before pushing a screen is a property of the handler body, and no diagnostic reports its absence. Timing is further from the type system still. The three empty compile-time cells in Fig. 2 are therefore structural. They will not fill with a future release of the checker, because the properties they represent are not properties of types.

The deterministic test mode has the opposite blind spot. It verifies the interleavings that a test constructs, and a test constructs only the interleavings that its author imagined. A lock-based cache that compiles under the Swift 5 language mode may serialize every access that the tests exercise and still expose a mutation on a path that no test drives. First-

hand experience with moving a codebase toward Sendable conformance and actor isolation is instructive on this point. The migration surfaced possible shared-state mutations that lock-based code had hidden from the compiler and, by the same token, from the test suite, because a lock that is acquired on every tested path conceals the path on which it is not acquired. The thread sanitizer narrows the blind spot without closing it, since it observes only the runs it instruments.

The two blind spots do not overlap. The properties the compiler cannot express are the properties that a deterministic test controls, and the paths a deterministic test cannot exercise are the paths that the compiler checks without exercising them. Both modes are required for asynchronous code that can be trusted, and neither is a substitute for the other. The gap that remains after both are in place is a code that admits neither mode. Closing that gap is a matter of design, and the next subsection derives the design from the matrix.

3.5 Structure of a Deterministic Test Environment Derived from the Matrix

The union of the cells that the run-time approaches fill in Table 3 defines what a test environment must be able to inject. Fig. 3 expresses the division of responsibility as a flow from a property of an asynchronous path to the mode that verifies it, and from each mode to its blind spot, and from both blind spots to the design that closes them. Four seams follow from the four run-time dimensions. The clock is injected as a protocol conforming value, so that every sleep, debounce, timeout, and delay in the component resolves against a time source the test advances [21]. The scheduler or executor is injected for the same reason, so that the order in which queued work runs is a decision of the test [22]. Dependencies are injected at the boundary of the module under test and at no deeper level, because a dependency replaced two layers down alters behavior that the test does not own and couples the test to the internal structure of the graph. Cancellation is made observable, either by an effect lifetime that the harness tracks or by a cancellation token the test can inspect, so that a request left in flight at the end of a test registers as a failure [23].

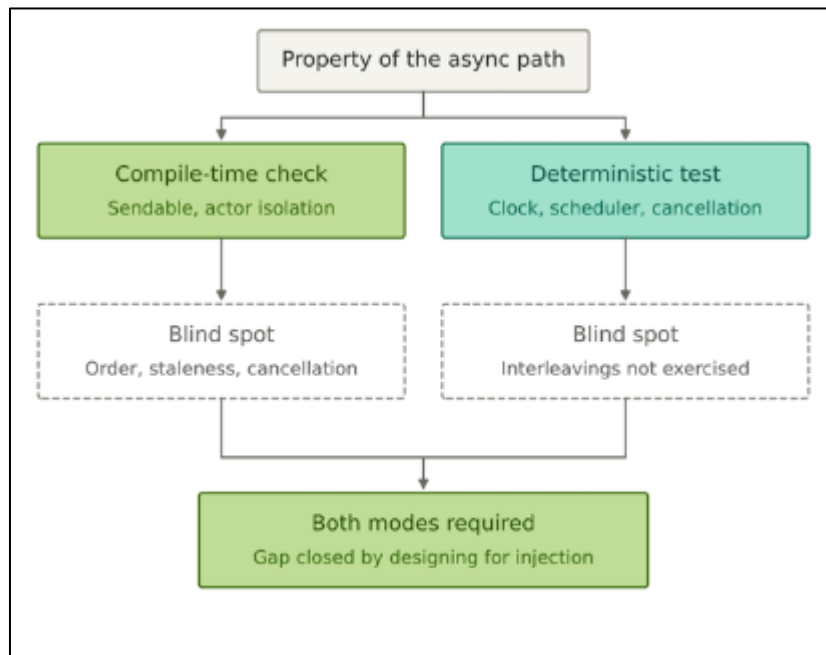


Figure 3: Division of responsibility between the two verification modes and the design that closes their blind spots

The test driver in such an environment advances the clock by explicit steps, runs pending work after each step, asserts on state between steps, and checks that every effect has completed or been cancelled at the end. No wall-clock time elapses, and the same sequence of steps produces the same sequence of states on a workstation and on a loaded runner. Direct experience with mock dispatch queues and with the composable architecture test clocks confirms that the construction is practical. The value of the construction lies in the ability to stop between steps and to assert on state before continuing, which is the capability that a sleep removes.

The compile-time mode contributes to the environment in a way that Fig. 3 records in its second tier. A component that is Sendable at its boundaries and actor-isolated in its state has, as a side effect of satisfying the checker, declared which values cross which boundaries, and a test that injects a seam at a declared boundary injects it where the compiler already expects a boundary to be. Strict concurrency therefore documents the seams that a deterministic test needs, and this is the reason Table 3 codes the Swift 6 language mode as partial for dependencies. The two modes meet at the module boundary. The compiler checks what crosses it, and the test controls when.

4 DISCUSSION

4.1 Why Waiting Persists

The persistence of waiting as a practice has causes that the review can name without measuring their weight. The first cause is the low cost of entry. A sleep is one line, requires no change to the component under test, and passes on the machine where it is written. A deterministic reproduction of the same behavior requires that the component accept a clock, that the clock be reachable from the test, and that the author understand which step of virtual time triggers which effect. The survey of developer experience with flaky tests records rerunning, quarantining, and adjusting the test among the actions developers take in response, and each of those actions leaves the nondeterminism in place [11]. The multivocal review of the field records rerun-based detection as the most common method and traces the same pattern through the grey literature, where retries and timeout adjustments appear as routine responses [2].

The second cause is that the cost of waiting is paid by the suite while the test that incurs it shows no sign of it. A sleep that lets a single test pass adds its full duration to every run of that test, and a suite in which each test sleeps for the maximum expected latency grows in duration in proportion to its size. A deterministic reproduction of the same tests runs in the time the assertions take, because no wall-clock time elapses between steps. The difference between the two suites is therefore a difference between a cost that is visible once at the moment of writing and a cost that is invisible and recurring, and teams that have not measured the second cost have no basis on which to justify the first.

4.2 Complementarity and What the Swift 6 Language Mode Changes

The central claim of the review is that the two modes are complementary along a boundary that the evidence draws with some precision. For a team that relied on deterministic tests alone, the Swift 6 language mode adds a guarantee that no test could provide, namely that no path in the module, exercised or not, mutates shared state across an isolation boundary without a diagnostic. For a team that relied on strict checking alone, the review supplies a list of the properties that the checker does not reach. The three empty compile-time cells in Fig. 2 name ordering and staleness, cancellation, and timing, and the first and fourth scenarios of Table 2 show that two of them produce a user-visible failure in code that compiles without warning, and the study of computational resources shows that the third produces a verdict that changes with the machine [4].

4.3 Threats to Validity

External validity is bounded by the origin of the scenarios. Four scenarios from first-hand experience with a small number of codebases cannot represent the population of iOS applications, and the review makes no such claim. Their standing rests on their correspondence with documented counterparts in user interface flakiness research and in the Kotlin coroutine study [12,15]. Conclusion validity is limited by the absence of measurement. The review states which properties each mode reaches and does not state how often each blind spot produces a defect in practice.

5 CONCLUSION

Compile-time checking and deterministic run-time testing solve different parts of one problem, and asynchronous iOS code that can be trusted requires both. The compiler, through Sendable conformance and actor isolation, guarantees the absence of memory-level data races on every path in a module, and it expresses nothing about the order in which asynchronous operations complete, about whether in-flight work is cancelled, or about the passage of time. A deterministic test, through an injected clock, an injected scheduler, dependencies replaced at the module boundary, and observable cancellation, controls those three properties along the interleavings it constructs and reaches no interleaving it does not construct. The two blind spots do not overlap, and the union of the two modes is the coverage that neither reaches alone.

The practice of waiting, whether as a sleep or as an expectation with a timeout, controls none of the five dimensions and adds delay to the suite while leaving the failure it was written to avoid available on any slower machine. The four scenarios that motivate the review, stale overwrite, competing updates, read before completion, and navigation after a missing cancellation, each become deterministic under the environment derived here, in which a test advances virtual time by explicit steps and asserts on state between them. The remaining gap is code that admits neither mode, and it closes through design for injection at the module boundary, where the seam that the compiler already requires is the seam that the test already needs.

Future work should replace the analytical coding with an empirical one. A study of open-source iOS repositories that records the adoption of strict concurrency, the presence of injected clocks and schedulers, and the repair history of timing-dependent tests would test the claim that the two modes are adopted as substitutes, and it would measure the cost of the waiting practice that this review could only observe.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Disclosure of conflict of interest

No conflict of interest to be disclosed.

REFERENCES

- [1] Parry O, Kapfhammer GM, Hilton M, McMinn P. A survey of flaky tests. *ACM Trans Softw Eng Methodol.* 2021;31(1):1–74. doi:10.1145/3476105
- [2] Tahir A, Rasheed S, Dietrich J, Hashemi N, Zhang L. Test flakiness' causes, detection, impact and responses: a multivocal review. *J Syst Softw.* 2023;206:111837. doi:10.1016/j.jss.2023.111837
- [3] Rahman S, Shi A. FlakeSync: automatically repairing async flaky tests. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024; 2024 Apr 14–20; Lisbon, Portugal.* New York: ACM; 2024. Article 136, p. 1–12. doi:10.1145/3597503.3639115
- [4] Silva D, Gruber M, Gokhale S, Arteca E, Turcotte A, d'Amorim M, et al. The effects of computational resources on flaky tests. *IEEE Trans Softw Eng.* 2024;50. doi:10.1109/TSE.2024.3462251
- [5] Swift Evolution. SE-0296: async/await [Internet]. 2021 [cited 2026 Sep 9]. Available from: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0296-async-await.md>
- [6] Swift Evolution. SE-0306: actors [Internet]. 2021 [cited 2026 Sep 9]. Available from: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0306-actors.md>
- [7] Swift Evolution. SE-0302: Sendable and @Sendable closures [Internet]. 2021 [cited 2026 Sep 9]. Available from: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0302-concurrent-value-and-concurrent-closures.md>
- [8] Swift.org. Migrating to Swift 6: Swift 6 language mode migration guide [Internet]. 2024 [cited 2026 Sep 9]. Available from: <https://www.swift.org/migration/documentation/migrationguide/>
- [9] Kubiak T. From UIKit to SwiftUI: a reproducible protocol and a priori typology for open-source iOS application migration. *Am Impact Rev.* 2026:e2026068. doi:10.66308/air.e2026068
- [10] Kubiak T. A local-first architecture for privacy-preserving personalization in iOS health and fitness applications. *Am Impact Rev.* 2026:e2026058. doi:10.66308/air.e2026058
- [11] Parry O, Kapfhammer GM, Hilton M, McMinn P. Surveying the developer experience of flaky tests. In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2022; 2022 May 21–29; Pittsburgh, PA, USA.* New York: ACM; 2022. p. 253–62. doi:10.1145/3510457.3513037
- [12] Romano A, Song Z, Grandhi S, Yang W, Wang W. An empirical analysis of UI-based flaky tests. In: *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021; 2021 May 22–30; Madrid, Spain.* Piscataway, NJ: IEEE; 2021. p. 1585–97. doi:10.1109/ICSE43902.2021.00141
- [13] Dong Z, Tiwari A, Yu XL, Roychoudhury A. Flaky test detection in Android via event order exploration. In: *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021; 2021 Aug 23–28; Athens, Greece.* New York: ACM; 2021. p. 367–78. doi:10.1145/3468264.3468584
- [14] Milano M, Turcotti J, Myers AC. A flexible type system for fearless concurrency. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022; 2022 Jun 13–17; San Diego, CA, USA.* New York: ACM; 2022. p. 458–73. doi:10.1145/3519939.3523443
- [15] Brockbernd B, Koval N, van Deursen A, Ozkan BK. Understanding concurrency bugs in real-world programs with Kotlin coroutines. In: Aldrich J, Salvaneschi G, editors. *38th European Conference on Object-Oriented Programming, ECOOP 2024.* Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2024. p. 8:1–8:20. (Leibniz International Proceedings in Informatics; vol. 313). doi:10.4230/LIPIcs.ECOOP.2024.8
- [16] Ibañez Kautsch A. *Modern concurrency on Apple platforms: using async/await with Swift.* Berkeley, CA: Apress; 2023. doi:10.1007/978-1-4842-8695-1
- [17] Swift Evolution. SE-0304: structured concurrency [Internet]. 2021 [cited 2026 Sep 9]. Available from: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0304-structured-concurrency.md>

- [18] Swift Evolution. SE-0337: incremental migration to concurrency checking [Internet]. 2022 [cited 2026 Sep 9]. Available from: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0337-support-incremental-migration-to-concurrency-checking.md>
- [19] Swift Evolution. SE-0414: region based isolation [Internet]. 2024 [cited 2026 Sep 9]. Available from: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0414-region-based-isolation.md>
- [20] Apple. Eliminate data races using Swift Concurrency. WWDC22 session 110351 [Internet]. 2022 [cited 2026 Sep 9]. Available from: <https://developer.apple.com/videos/play/wwdc2022/110351/>
- [21] Point-Free. swift-clocks: a few clocks that make working with Swift concurrency more testable and more versatile [Internet]. GitHub; 2022 [cited 2026 Sep 9]. Available from: <https://github.com/pointfreeco/swift-clocks>
- [22] Point-Free. combine-schedulers: a few schedulers that make working with Combine more testable and more versatile [Internet]. GitHub; 2020 [cited 2026 Sep 9]. Available from: <https://github.com/pointfreeco/combine-schedulers>
- [23] Point-Free. The Composable Architecture: TestStore documentation [Internet]. GitHub; 2020 [cited 2026 Sep 9]. Available from: <https://github.com/pointfreeco/swift-composable-architecture>
- [24] Apple. XCTestExpectation. XCTest developer documentation [Internet]. Cupertino, CA: Apple; [cited 2026 Sep 9]. Available from: <https://developer.apple.com/documentation/xctest/xctestexpectation>
- [25] Apple. Swift Testing. Developer documentation [Internet]. Cupertino, CA: Apple; 2024 [cited 2026 Sep 9]. Available from: <https://developer.apple.com/documentation/testing>
- [26] Apple. Diagnosing memory, thread, and crash issues early. Xcode developer documentation [Internet]. Cupertino, CA: Apple; [cited 2026 Sep 9]. Available from: <https://developer.apple.com/documentation/xcode/diagnosing-memory-thread-and-crash-issues-early>