Austrian Journal of Technical
and Natural Sciences, 2026 No 7–8
ISSN 2310-5607

PREMIER Publishing
ppublishing.org

Section 2. Computer science

DOI:10.29013/AJT-26-7.8-58-66

TEST AUTOMATION IN IOS DEVELOPMENT: IMPLEMENTATION PRACTICES AND THEIR IMPACT ON PRODUCTION RELEASE STABILITY

Daniil Nuzhdin [1]

[1] Lomonosov Moscow State University, Moscow, Russia

Cite: *Nuzhdin D. (2026). Test automation in iOS development: implementation practices and their impact on production release stability. Austrian Journal of Technical and Natural Sciences 2026, No 7–8.* https://doi.org/10.29013/AJT-26-7.8-58-66

Abstract

The article examines current approaches to test automation in iOS development and their influence on production release stability. It is shown that automation extends beyond an auxiliary QA practice and becomes an element of a team's engineering maturity, affecting delivery predictability, release speed, and the reduction of regression risks. The study analyzes the levels of automated testing, criteria for tool selection, the integration of checks into the CI/CD pipeline, and the use of automation as a mechanism for release admission. Based on industry benchmarks and engineering cases, it is argued that the greatest effect is achieved through the systematic integration of automated tests into the release process and the evaluation of their effectiveness through post-release quality metrics.

Keywords: *test automation, iOS development, CI/CD, production releases, application stability, mobile testing, engineering maturity*

Introduction

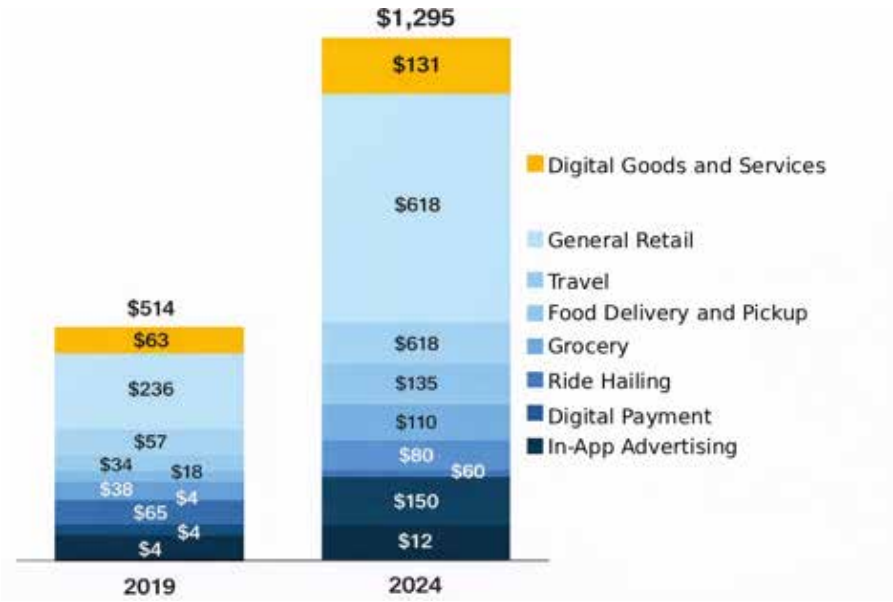
The development of mobile applications for the iOS ecosystem imposes high requirements on software quality, release cycle stability, and the predictability of application behavior in the production environment. The increasing complexity of mobile solution architectures, the growing number of integrations, the higher frequency of updates, and rising user expectations make it insufficient to rely exclusively on manual verification procedures. Under these conditions, test automation becomes not an auxiliary but a system-forming element of the engineering process, making it possible to detect defects at early stages of development, ensure the reproducibility of checks, and maintain application stability under continuous change. This approach is of particular importance in iOS development, where requirements for performance, interface consistency, and the reliability of interaction with platform components directly affect user experience and the quality of production releases.

The relevance of the study is determined by the need for a systematic analysis of automated testing implementation practices throughout the lifecycle of an iOS application and by the need to assess their influence on the stability of the release process. Despite the widespread adoption of automation tools, their effective use requires a well-grounded selection of testing levels, an appropriate choice of frameworks, integration into CI/CD pipelines, and consideration of the organizational characteristics of the development team. The aim of the article is to analyze current approaches to test automation in iOS development, identify practices of its implementation at different stages of the application lifecycle, and determine the impact of automated testing on release speed, defect reduction, and the stability of production releases.

Test automation as an indicator of engineering maturity in iOS development teams

At the current stage of iOS development, test automation should be regarded not as a local quality control tool, but as one of the key indicators of a team's engineering maturity (Singh & Jain, 2025). This is driven not only by the growing complexity of mobile application architectures, but also by the scale of the iOS ecosystem itself. According to the latest Analysis Group study supported by Apple, in 2024 the App Store ecosystem generated approximately $1,295 billion in global billings and sales, while App Store applications were visited by more than 813 million users per week on average (fig. 1).

Figure 1. *Estimated billings and sales facilitated by the App Store ecosystem by app category, 2019 to 2024, billion dollars (Burley & Fradkin, 2025)*



In such an environment, even isolated regressions reaching production carry substantial operational and reputational costs, since release quality affects not only user satisfaction, but also a product's competitiveness in a highly saturated digital market.

From an engineering perspective, iOS team maturity is reflected in the ability to transform testing from a final verification stage into a **continuous mechanism for validating changes.** According to the modern DORA methodology, there is an evident correlation between the speed and stability of software delivery performance and certain throughput and stability measures such as lead time for change, deployment frequency, time to restore service, change fail rate, and deployment rework rate. More importantly, DORA has established that speed and stability need not necessarily be incompatible. In iOS development, this means that automated unit, integration, UI, and performance tests must be embedded into the release pipeline to increase delivery frequency while reducing the risk of failed deployments (Harvey, 2026).

The **Mobile App Stability Outlook 2025 report** can be used as an empirical benchmark for assessing mobile release stability requirements. The study analyzed data from mobile applications that used the Instabug platform throughout 2024 to monitor stability and user experience quality. Its key metrics included crash-free sessions as well as additional reliability indicators such as ANR, OOM, app hangs, and forced restarts. This approach makes it possible to interpret stability not as an abstract quality attribute, but as a measurable operational indicator directly linked to engineering maturity and a team's ability to keep production releases within a competitive stability standard (table 1).

Table 1. *Benchmark indicators of mobile application stability based on the Mobile App Stability Outlook 2025 report (Luciq, 2025)*

Metric	Value	Analytical relevance in the context of the article
Median crash-free sessions rate	99.95%	Baseline benchmark for acceptable mobile app stability.
75th percentile crash-free sessions rate	99.99%	Benchmark of high-performing mobile teams.
Median iOS crash-free sessions rate	99.91%	Indicates a high stability standard for iOS release quality.
Median Android crash-free sessions rate	99.80%	Provides a comparative benchmark showing lower median stability than iOS.

The benchmark data show that mobile release stability has become a measurable engineering target rather than a purely qualitative characteristic. For iOS teams, maintaining crash-free session rates close to the upper benchmark range can be treated as an indicator of a mature test automation strategy and a reliable release process. In this context, automated testing should be viewed not only as a verification tool, but also as a practical mechanism for achieving and sustaining competitive production stability.

Apple's tooling further reinforces this logic. **Xcode Cloud** includes built-in CI/CD capabilities for Apple development that allow for parallel automated test runs, instant feedback on each commit, and automatic aggregation of build, test, and crash reporting information in Xcode. On the other hand, Swift Testing in Xcode 16 adds support for parameterized tests, automatic parallelization of tests, and interoperability with XCTest. This makes modernizing the testing process step by step much more feasible. Consequently, mature organizations will be able to create a continuous validation process where lightweight checks will be validated instantly, whereas more resource-intensive scenarios will be executed in a CI/CD pipeline.

Importantly, the expansion of automation does not necessarily slow mobile delivery, even as the number of quality gates increases. According to **the report Bitrise Mobile Insights 2025**, the level of complexity in mobile CI pipelines went up by 23%, but at the same time, the best teams have reduced build times by 28% in the last 3.5 years on the basis of analysis of over 10 million builds. In addition, in another report released in March 2026, which was based on an analysis of tens of millions of builds done from 2022 to 2025, the authors suggested that teams had been able to introduce additional steps into the build process aimed at testing, scanning, and verifying code-signing without any reduction of speed. In this regard, it can be stated that the maturity of iOS engineering is more defined by its automated delivery pipeline than by the number of performed tests since it will reduce the duration of bug fixing and recovery from failed builds while maintaining the production stability.

In other words, the automation of iOS testing is not only a technical approach but also a critical factor of engineering maturity that plays an important role beyond just finding defects.

Multi-level organization of automated testing in iOS development

The effectiveness of automated testing in iOS development is determined not only by the set of tools used, but also by how testing activities are distributed across levels and integrated into the release pipeline. In practice, this requires a clear distinction between the functions of unit, integration, UI, performance, and pre-release testing, since each level addresses a specific class of risks and plays a separate role in ensuring application stability (table 2).

Table 2. *Levels of automated testing in iOS development and their functional role (Sharma, 2025; Afrihyia et al., 2022)*

Testing level	Primary purpose	Typical objects of verification	Role in the release pipeline
Unit testing	Verification of business logic correctness and local computations.	Models, services, error handling, data transformation.	Provides the fastest feedback at the code level.
Integration testing	Verification of interactions between application components.	Network adapters, persistence layer, service abstractions, inter-module connections.	Helps identify integration errors before they reach the user-facing layer.
UI testing	Validation of user scenarios and navigation flows.	Screens, transitions, forms, system interruptions, interface elements.	Confirms the correctness of critical user journeys.
Performance testing	Assessment of execution performance and runtime stability.	Response time, resource usage, launch speed, behavior under load.	Reduces the risk of user experience degradation after release.
Pre-release beta validation	Verification of build behavior under conditions close to real use.	TestFlight builds, real devices, network scenarios, rare configurations.	Complements the CI/CD pipeline and reduces the risk of production defects.

The presented structure shows that automated testing in iOS development should be understood as a multi-level system of engineering control rather than as a set of isolated checks. Its maturity is determined by the team's ability to distribute test scenarios rationally across levels, combine fast feedback-oriented checks with more expensive user-facing tests, and complement the server-side CI/CD pipeline with pre-release validation on real configurations. It is this organization that makes it possible to maintain delivery speed, reduce the probability of regressions, and ensure the stability of production releases.

Within this multi-level structure, performance testing requires separate attention because performance risks in iOS applications are not limited to response time and often appear only at runtime, under realistic interaction patterns and device constraints. For iOS teams, performance testing should go beyond a general check of response time. The problems that most often affect a release are usually more specific: slow cold starts, delayed first-screen rendering, dropped frames while scrolling collection views, unexpected main-thread work, memory growth after repeated user flows, leaks, out-of-memory crashes, app hangs, and excessive energy consumption. These risks become particularly noticeable in screens with large lists, images, animations, media processing, maps, or a mixed SwiftUI and UIKit hierarchy.

In practice, an effective testing setup can combine XCTest performance measurements, Instruments, MetricKit, Xcode Organizer, and production crash or hang reports. This approach helps teams detect not only functional failures, but also performance regressions that make the application feel slower or less stable even when all user scenarios still pass. Therefore, the selection of testing tools should be connected not only with the type of test being implemented, but also with the specific risk that each level is expected to control.

In actual iOS app development, the choice of testing tools is determined not only by technical preferences, but also by the trade-off between execution speed, maintenance costs, integration with the iOS ecosystem, and the risks that need to be mitigated. Usually, XCTest and Swift Testing are used in situations where developers need fast, deep-level integration testing, while XCUITest works better in cases of testing important user interactions from an interface perspective. It is equally important to pay attention to how frequently the tested level will change, how sensitive testing will be to UI instabilities, and what maintenance costs will be associated with maintaining test suites. Consequently, good automation practice in iOS does not involve using one framework, but rather aligning them with their architectural purpose within the testing hierarchy.

However, the effectiveness of this testing hierarchy depends not only on the selected tools, but also on the architectural testability of the application. Testability starts with architecture rather than with the testing framework itself. MVVM, VIPER, TCA, and Clean Architecture are useful only when they actually move state, decisions, navigation, and side effects out of UIViewController and SwiftUI View. If a view controller contains validation rules, networking decisions, analytics logic, and navigation branching, automated tests become slow and fragile because the test has to deal with lifecycle, rendering, and platform behavior before it can check the business rule. When the same logic is placed in a view model, interactor, reducer, use case, or domain service, it can be tested quickly and deterministically without launching the app.

The dependency injection technique allows for making this separation practical. Network clients, storage, analytics, feature flags, authorization state, clock, scheduler, and configuration providers should be injected into classes through protocols or abstractions so that they can be substituted during testing. Coordinators and routers can also be tested independently from the full user interface flow: it becomes possible to verify which route was called, which screen should be opened, or which navigation command was created. For async/await, Combine, and Observation-based code, the important part is to control timing, cancellation, emitted values, and state transitions instead of relying on arbitrary waits. The same principle applies to mocks for network, persistence, and analytics: they should live on stable boundaries and be reviewed when backend contracts change. A modular project structure helps here as well, because domain and service modules can have small, fast test targets, while UIKit, SwiftUI, and full application targets are left for fewer, higher-level checks.

Practices for implementing test automation and their impact on production release stability

The practical impact of automation on production release stability is especially evident when **pre-release checks that were previously performed manually are automated** (Mulla, 2024). A representative example is **GetYourGuide**, where the phased introduction of automated release checks based on XCUITest and KIF, their integration into CI, and their scaling across mobile teams made it possible to substantially reduce the volume of manual pre-release work and shorten the release window (table 3).

Table 3. *Quantitative results of implementing automated release testing in GetYourGuide's mobile development process (Oliveira, 2024)*

Metric	Before automation	After automation	Analytical relevance in the context of the article
Number of manual release tests	46	4	Shows that automation enabled most release-critical checks to be transferred from manual execution to a reproducible automated workflow.

Metric	Before automation	After automation	Analytical relevance in the context of the article
Number of engineers involved in manual pre-release validation	7	4	Reflects reduced labor intensity at the final delivery stage and lower dependence on manual release coordination.
Share of automated release tests	0%	about 50% by early 2022	Characterizes the phased nature of implementation and shows that even partial automation can generate a measurable operational effect.
Interval from code freeze to release	about 7 days	about 2 days	Demonstrates the reduction of the most vulnerable pre-release period, where delays and regression risks typically accumulate.
Maximum time from code submission to user availability	up to 3 weeks	significantly reduced	Confirms that automation affects not only local testing procedures but also the overall predictability of mobile delivery.

The presented data show that the effect of automation in mobile development is determined not only by the acceleration of individual testing operations, but also by the restructuring of the entire pre-release workflow. A reduction in the number of manual release checks, a shorter interval between code freeze and release, and a lower workload on engineers help decrease the probability of defect accumulation at the final stage and improve the stability of the production release. In this sense, automation acts not merely as a means of speeding up testing, but as a tool for reducing release risks and increasing the controllability of delivery quality.

A second important practice is the **transformation of automated tests from a supporting verification mechanism into a formal release admission criterion**. For example, Spotify's mobile release process operates on a weekly cadence, but a build does not move forward until all commits in the release branch are included in the current build, pass automated tests, and remain within predefined crash-rate and other quality thresholds. Rollout is then performed in two phases: first to 1% of users and, if no negative signals appear, to 100% the following day. With only 3–5 release-blocking bugs typically identified per release and fixed before publication, this case shows that automation can function not only as a development accelerator, but as a core element of release stabilization through the integration of automated tests, operational metrics, and phased deployment.

A third important practice is the **systematic detection and management of flaky tests as a prerequisite for stable releases**. Uber's engineering experience shows that this problem becomes critical at scale: its CI infrastructure validates more than 2,500 code changes per day and runs over 10,000 tests per diff on average. After introducing a centralized flaky-test management system, Uber reported the steady detection of about 1,000 flaky tests out of 600,000 in the Go monorepo and about 1,000 out of 350,000 in Java, together with improved CI reliability and a substantial reduction in retries. This case demonstrates that, in mature automation practice, production stability depends not only on functional test coverage, but also on the ability to identify unstable tests early, isolate their impact on CI, and enforce accountability for their resolution. In addition, recent studies indicate that the automation of testing and debugging tasks is increasingly supported by intelligent tools, including large language models, which may further expand the capabilities of modern software delivery environments (Mozharovskii et al., 2024).

From an operational perspective, the value of test automation is fully realized only when it is linked to post-release quality metrics. In iOS teams, this means that the effectiveness of automation should be assessed not only through coverage or CI

duration, but also through change fail rate, rollback frequency, crash-free session rate, hotfix volume, and the number of production incidents after release. Such a measurement approach makes it possible to evaluate automation not as an internal engineering artifact, but as a delivery mechanism with observable effects on release stability. As a result, the maturity of an automation strategy can be judged by how consistently it improves both pre-release confidence and post-release performance.

Taken together, these cases show that the stabilizing effect of test automation is achieved not through the mere expansion of test coverage, but through its targeted integration into the release path. Automation has the greatest impact when it reduces manual pre-release effort, serves as a formal release gate, and prevents unstable tests from degrading CI reliability. In this sense, production release stability in iOS development depends on how effectively automated testing is embedded into release management, quality control, and the operational discipline of delivery.

A practical rollout of automation should therefore be gradual and tied to actual release risks. At the initial stage, business logic should be covered with fast unit tests, including computations, validation rules, data mapping, state transitions, and error handling. The next stage should focus on critical services and scenarios, such as authorization, payments, caching, data migration, token refresh, offline behavior, and backend error handling. After this foundation has been established, smoke UI tests can be introduced for flows whose failure would block a release, including login, onboarding, purchase, payment, profile management, and the main product scenario. Snapshot or visual regression tests may additionally be used for screens where visual correctness is important and manual verification is repetitive. CI gates should also be introduced gradually: fast checks on each pull request, heavier suites before release or during scheduled nightly runs, and a separate policy for quarantining flaky tests with an assigned owner and a fixed deadline for correction or removal. Finally, the effectiveness of automation should be assessed through product-level outcomes, including crash-free sessions, hotfix frequency, regression bugs, rollback frequency, and production incidents.

Limitations and implementation constraints of test automation in iOS development

Despite the significant practical value of automated testing, its implementation in iOS development is associated with a number of technical and organizational constraints. In practice, the effectiveness of automation depends not only on the choice of tools, but also on the cost of maintaining test suites, the stability of CI infrastructure, the sensitivity of UI scenarios to interface changes, and the availability of engineering resources required to keep the testing pipeline reliable over time. For this reason, the impact of automation on production stability should be assessed with regard not only to its benefits, but also to the limitations that may reduce its operational efficiency (table 4).

Table 4. *Key limitations of automated testing implementation in iOS development (Prodduturi, 2025; Singh, 2025)*

Limitation	Practical manifestation	Potential effect on release stability
High maintenance cost of UI tests	UI scenarios often require frequent updates after interface or navigation changes.	Can reduce the reliability of release checks and increase the cost of sustaining the automation suite.
Flaky test behavior	Unstable results in repeated runs create false positives and reduce trust in CI outcomes.	May delay releases and mask real regressions.

Limitation	Practical manifestation	Potential effect on release stability
Infrastructure dependence	Automated testing in iOS requires stable macOS-based CI environments, device availability, and code-signing consistency.	Failures in infrastructure can affect the reproducibility of release validation.
Uneven test coverage across levels	Teams often automate code-level checks earlier than complex end-to-end or real-device scenarios.	Critical production issues may still escape into release despite high formal test activity.
Resource and expertise requirements	Effective automation requires engineering time, ownership, and continuous monitoring of test quality.	Weak operational discipline reduces the long-term stabilizing effect of automation.

The limitations summarized in table 4 show that the main challenge is not the initial creation of automated tests, but the preservation of their practical value under continuous product change. UI tests may become unstable after interface redesigns, long test suites may slow down CI feedback, and repeated false failures may reduce developers' trust in the pipeline. Test doubles also require regular updating: if backend contracts, feature flags, or analytics requirements change while mocks remain unchanged, tests may continue to pass even when the real application behavior is already incorrect. Flaky tests require explicit ownership; otherwise, they increase retries, create background noise, and obscure real regressions. Another common problem is coverage created for reporting rather than for risk reduction: such tests improve formal metrics but do not protect the scenarios that are most important for users and business continuity. For this reason, mature iOS teams should regularly review their test suites, remove low-value checks, separate slow scenarios from the pull-request pipeline, and keep automation focused on release confidence rather than on formal indicators.

These constraints show that test automation should not be treated as a self-sufficient guarantee of release quality. Its contribution to production stability depends on whether the team is able to maintain test suites, control flaky behavior, support the necessary infrastructure, and align automation depth with the real risk profile of the product. In this sense, the maturity of an iOS automation strategy is determined not only by what is automated, but also by how sustainably the entire verification system can be operated under continuous product change.

Conclusion

The analysis shows that test automation in iOS development should be understood as a systemic engineering practice that affects the entire release lifecycle rather than as a limited QA procedure. Its practical value is determined by the ability to distribute validation across multiple testing levels, integrate checks into CI/CD workflows, reduce manual pre-release effort, and use automation as a release control mechanism. The reviewed cases and benchmarks indicate that automation contributes to production stability not only through earlier defect detection, but also through shorter release windows, more predictable delivery, and a lower probability of critical regressions reaching production.

At the same time, the effectiveness of automation depends not on the formal growth of coverage, but on how precisely testing practices are aligned with product risks, release workflows, and operational quality metrics. The strongest effect is achieved when automation supports release admission decisions, stabilizes CI reliability, and is evaluated through measurable post-release outcomes such as crash-free session rate, rollback frequency, hotfix volume, and production incident count. Therefore, the long-term significance of automated testing in iOS development lies in its ability to balance release velocity with production resilience under conditions of continuous product change.

References

Afrihyia, E., Umana, A. U., Appoh, M., Frempong, D., Akinboboye, O., Okoli, I., Umar, M. O., & Omolayo, O. (2022). Enhancing software reliability through automated testing strategies and frameworks in cross-platform digital application environments. Journal of Frontiers in Multidisciplinary Research, – 3 (2). – P. 517–531. URL: https://doi.org/10.54660/.JFMR.2022.3.1.517–531

Burley, J., & Fradkin, A. (2025). The Global App Store and Its Growth. Apple. [cited: 10.04.2026]. Available from: https://www.apple.com/newsroom/pdfs/2024-Apple-Global-Ecosystem-Report-June2025.pdf

Harvey, N. (2026). DORA's software delivery performance metrics. DORA. [cited: 10.04.2026]. Available from: https://dora.dev/guides/dora-metrics/

Luciq. (2025) Mobile app stability outlook 2025. [cited: 10.04.2026]. Available from: https://www.luciq.ai/mobile-app-stability-outlook-2025

Mozharovskii, E., Aluev, A., Dudak, A., & Ishankhonov, A. (2024). Application of large language models for software testing and debugging automation. Modern Science: Actual Problems of Theory and Practice. Series "Natural and Technical Sciences", 11–2, 103–108. URL: https://doi.org/10.37882/2223–2966.2024.11–2.22

Mulla, F. A. (2024). Modern Mobile Testing Tools: A Comprehensive Guide to Quality Assurance and Automation. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, – 10 (6). – P. 1577–1584. URL: https://doi.org/10.32628/CSEIT241061203

Oliveira, F. (2024). The Long and Winding Road to Short and Smooth Releases. GetYourGuide. [cited: 11.04.2026]. Available from: https://www.getyourguide.careers/posts/the-long-and-winding-road-to-short-and-smooth-releases

Prodduturi, S. M. K. (2025). Efficient Debugging Methods and Tools for iOS Applications Using Xcode. International Journal of Data Science and IoT Management System, – 4 (4). – P. 1–6. URL: https://doi.org/10.64751/ijdim.2025.v4.n4.pp1-6

Sharma, A. (2025). Security Testing in Android and iOS Application Development. Journal of Android, iOS Development and Testing, – 10 (2). – P. 96–109.

Singh, S., & Jain, A. (2025). Automated Testing Strategies for Android and iOS Applications. Journal of Android, iOS Development and Testing, – 10 (2). – P. 55–69.

Singh, V. (2025). Advancements in Software Engineering Practices and Their Influence on Modern Software Testing Methodologies: A Comprehensive Study. Journal of Software Engineering & Software Testing, – 10 (2). – P. 71–83.

submitted 27.06.2026;
accepted for publication 11.07.2026;
published 31.08.2026

Contact: daniila.nuzhdin@yandex.ru