American Academic Scientific Research Journal for Engineering, Technology, and Sciences
ISSN (Print) 2313-4410, ISSN (Online) 2313-4402
https://asrjetsjournal.org/index.php/American_Scientific_Journal/index

Interaction between IDEs and AI Assistants in the Process of Writing Automated Tests

Vladyslav Korol*

Software Developer Engineer In Test (Penn Entertainment), USA, Miami
Email: vlad.korol@penn-interactive.com

Abstract

This paper looks at how modern, integrated development environments (IDEs) work with AI helpers to make it easier to write automated tests. The paper tries to see how far these kinds of tools take away the boring workload from engineers and better the quality of test thinking. Fast growth in using AI helpers in IDEs justifies this study: 74% of developers plan on still using ChatGPT, and 41% — GitHub Copilot. More than 80% of firms have put Copilot into everyday use, and 90% say it has raised job happiness. Meanwhile, only 31% of suggestions are taken up, and 17% of those stay in the codebase, showing that AI has a small degree of independence on automated tests. This work remains novel in a systematic analysis that intertwines quantitative survey data with Copilot usage telemetry, and crowdsourced info on mobile platform fragmentation, further coupled with an architectural review of AI-plugin integration within Android Studio and IntelliJ IDEA. The support covers Kotlin Multiplatform, Gradle scripts, Page Object, Gherkin, and cross-platform steps generation. Suggestion acceptance rates, flakiness metrics, and CI/CD licensing requirements have been integrated to evaluate the real-world impact of AI assistants. Findings at a high level indicate that helpful AI assistants in IDEs reduce the time it takes to build automated test scaffolds; helpful ones further automate locator choice considering Android/iOS fragmentation; they generate cross-platform scenarios in KMP projects and reduce flaky testing percentages through helpful wait-template recommendations. In return, though, suggestions are accepted by only about one-third of engineers; responsibility for quality and security of tests remains with the expert and therefore requires code review, static analysis, as well as license scanning. This article will help test developers, QA engineers, and automation architects who use AI helpers to make mobile and cross-platform testing better and more trustworthy.

Keywords: IDE; AI assistant; automated tests; GitHub Copilot; Kotlin Multiplatform; Page Object; mobile automation; flakiness; feature-flag; E2E-testing.

Received: 6/10/2025
Accepted: 8/1/2025
Published: 8/10/2025

---

* *Corresponding author.*

1. Introduction

Over the past two years, IDEs have evolved from code editors into innovative workstations: embedded helpers like GitHub Copilot scan project surroundings and suggest whole pieces of test logic. As per the Stack Overflow 2024 survey, 74% of developers plan to keep using ChatGPT, and 41% intend to add or keep Copilot in their workflow, making AI-driven code writing the new industry norm [1]. A corporate study by GitHub and Accenture showed that over 80% of participants have successfully integrated Copilot into daily practice, with 90% reporting increased job satisfaction [2]. Thus, the widespread adoption of AI assistants is no longer hypothetical but a statistically confirmed fact.

However, the effectiveness of such tools is still far from autopilot. Opsera telemetry analysis for April 2025 indicates that developers accept approximately 31% of Copilot suggestions, and only 17% of these remain in the final codebase after review [3]. These figures demonstrate that AI can relieve the tedium of generating templates. However, the final decision still belongs to the expert engineer, especially in automated tests, where the cost of an error is measured in defects slipped into production.

The inherent nature of mobile platforms adds further complexity. In BrowserStack's cloud farms, many iOS and Android devices with various OS versions, screen resolutions, and chipsets are available [4]. In contrast, it suffices to test a couple of dozen browser and OS combinations for a web application. Moreover, the iOS ecosystem updates more rapidly: by January 2025, 68% of all iPhones had migrated to iOS 18 [5]. Android lacks such convergence, requiring testers to account for dozens of API levels, various OEM customizations, and hardware sensors. Finally, mobile E2E tests must handle permission systems, power-saving modes, unstable radio channels, and asynchronous UI patterns, significantly complicating scenarios compared to the browser DOM.

These factors explain why demand for AI helpers in mobile automation is growing faster than in regular web testing: Without an auto code maker and IDE ideas, builders spend way more time setting up infrastructure, finding parts, and keeping cross-platform working. The following sections will show how Copilot and other models lower this wall by making routine steps easy and giving back time for risk checks and auto test setup.

2. Materials and Methodology

The research methodology includes different parts that work together using all 27 sources. To start, for measuring the number of AI helper acceptance, information from the Stack Overflow 2024 survey [1], JetBrains 2021 and 2023 surveys [8, 20], DevEcosystem 2024 [7], and GitHub/Accenture corporate study [2] was used. In these surveys, important metrics were measured—how often Copilot and ChatGPT are used, the rate at which their suggestions are accepted, and the subjective impact on productivity.

Secondly, to objectively assess the efficiency of automated test generation, we looked at Opsera telemetry for April 2025 [3] plus data on mobile platform split: iOS 18 market share from MacRumors [5] and Android version distribution from StatCounter [11]. To model test infrastructure, we checked device lists, BrowserStack cloud farms [4, 18], and ways to run tests simultaneously using Android Test Orchestrator and other similar

solutions[24].

Third, the qualitative review focuses on Copilot integration in IDEs and Multiplatform projects: plugin compatibility with Android Studio and IntelliJ IDEA [9], support for Kotlin Multiplatform and project structures commonTest/androidTest/iosTest [19, 21], generation of Page Object and Gherkin files based on Gradle script analysis [22]. Consideration of Android frameworks Espresso and UIAutomator [12–14] and iOS frameworks XCTest/XCUITest [15] is complemented by analysis of flakiness metrics from the Google Testing Blog [16, 17] and recommendations for wait-strategy management. Finally, legal and security issues were addressed based on Microsoft Copilot documentation [26] and CI/CD license-verification practices.

3. Results and Discussion

The transition from static autocomplete to generative AI began long before the advent of large language models: the first mass smart assistant, IntelliSense, appeared in Visual Studio as early as 1996, limited to suggesting names and function signatures without consideration of the broader project context [6]. Over a quarter of a century, algorithms matured from simple dictionary lookups to stochastic models capable of predicting entire code blocks. Between 2021 and 2025, this evolution accelerated exponentially thanks to Codex-like networks. As a result, in a global JetBrains survey [7], 40% of developers have already tried AI code generators, and 26% use GitHub Copilot regularly. ChatGPT has become a de facto reference: 69% of respondents apply it regularly. As shown in Fig. 1, most developers (67%) value AI tools primarily for reducing time spent searching for information, and over half note accelerated coding and development (58%), faster completion of repetitive tasks (57%), and overall productivity gains (57%). It has been reported by many users 45% that there is quicker onboarding with new technologies, and 39% said reduced cognitive load. Better code quality and overall development are noted much less often, 36% and 23%. Only a trivial percentage (1%) of the community, and that is negligible, sees no benefit from such tools, underscoring their broad demand and positive impact on workflows.

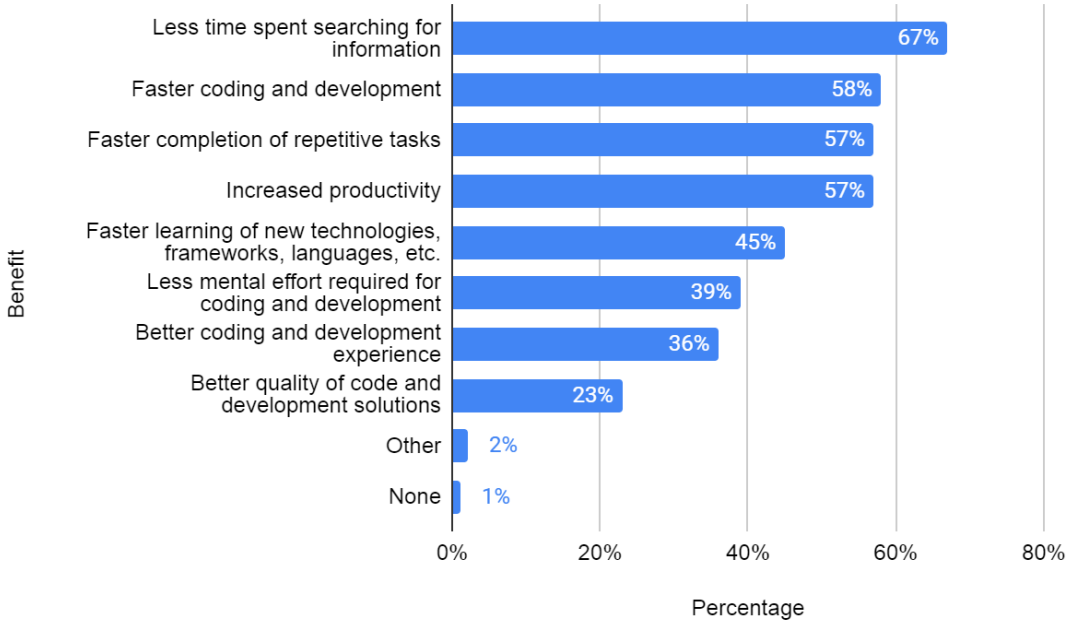

Figure 1: What benefits do you get from using AI tools for coding and other development-related activities? [7]

The evolution of tools has gone hand in hand with IDE infrastructure. For the Kotlin community, the core of the ecosystem remains two environments—Android Studio and IntelliJ IDEA. According to the DevEcosystem 2021 survey, 45% of Kotlin developers wrote code predominantly in Android Studio, 41% in IntelliJ IDEA Ultimate, and another 14% used the Community Edition; alternative editors like VS Code collectively accounted for only 1%, as shown in Fig. 2 [8].

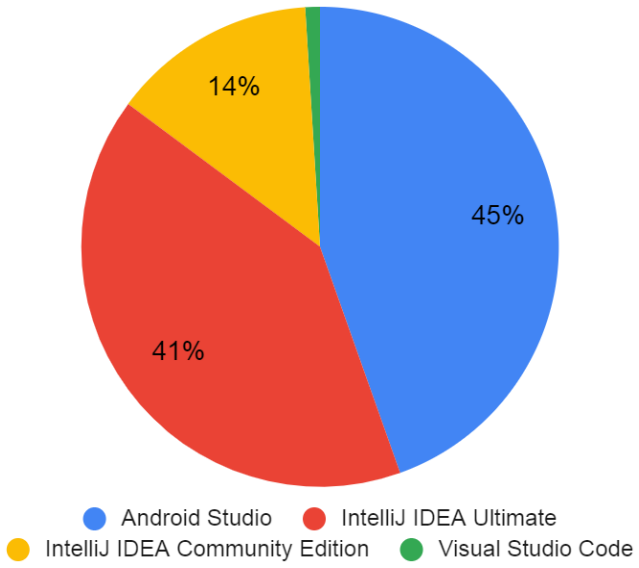

Figure 2: Which IDE do you use most regularly for Kotlin development? [8]

This usage distribution is explained by the tight integration of the Kotlin plugin with AST analysis, fast refactoring, and built-in testing tools, historically making JetBrains products the natural platform for advanced AI assistants. Microsoft positions Copilot as a first-class extension within the JetBrains family: the official plugin supports all IDE versions starting from the 2024.3 release, provides a unified authentication flow, and a full-featured chat interface within the editor [9]. This setup is essential for mobile automation testing in Kotlin: the helper notices Gradle scripts and Android/iOS module assets and can quickly propose Espresso or XCUITest steps while keeping the Page Object design. As a result, the coder gets an environment made of IDE + Copilot, where usual navigation and static checks of IntelliJ are added by likelihood-based advice, turning E2E test writing from a template-based task into a back-and-forth teamwork with the model.

The idea that an LLM could not only autocomplete a line but generate an entire end-to-end scenario ceased to be experimental when GitHub Copilot gained access to IDE telemetry and learned to identify recurring testing patterns. A joint GitHub–Accenture study covering 1,500 engineers from ten large companies found that on average, developers accept about one-third of Copilot's suggestions and retain 88 % of those after review [2]. These figures demonstrate that AI already offloads a significant share of routine work, although the final decision still rests with the engineer.

Classic scenarios in which Copilot brings benefit fall into three groups. Under the Page Object pattern it creates a page scaffold, ties locators to action methods, and inserts the correct framework imports; under the BDD approach the model unfolds a Gherkin file into a Kotlin Cucumber class, preserving the Given/When/Then structure; when migrating an existing test suite, Copilot can translate old JUnit 4 scripts into JUnit 5 or Kotest with minimal edits. Experience shows that time savings are especially noticeable in projects with large legacy codebases requiring the consolidation of test libraries. Understanding the project context is crucial: the plugin analyzes build.gradle.kts detects dependencies on Espresso or XCUITest, pulls in plugin versions, and automatically configures required Kapt generators. If a Kotlin Multiplatform build is declared, the assistant correctly distributes files into commonTest, androidTest, and iosTest folders, alleviating some cross-platform development routine.

A typical dialog with Copilot looks like this: the developer types into the chat, write a KMP step that authenticates the user via OAuth and verifies the presence of the Profile element on both platforms, and within seconds receives an AuthStep.kt file with an expect function login() and two actual implementations targeting Espresso and XCUITest. Such generations are beneficial where test steps are often repeated; the model quickly adapts to the internal repository and suggests contextual variants by the team's code style.

Copilot's greatest practical impact pairs with five UI automation tools on mobile platforms. For Android, the baseline choice remains Google Espresso: the espresso-core artifact is used in 153 public Maven Central projects [12]—more than twice as often as its nearest competitor, UIAutomator, which has 61 dependencies [13]. Beyond popularity, energy efficiency matters: in an extensive comparative study of eight mobile frameworks, maximum energy overhead reached 2200%, whereas Espresso exhibited the minimal overhead, earning recognition as the most energy-efficient choice for test farms [14]. When a test must interact outside its app—e.g., to confirm a system permissions dialog—Copilot automatically switches the template to

UIAutomator, preserving the page-object structure and importing the necessary driver classes.

For iOS, the assistant generates XCUITest code; in an industry survey published in April 2025, 85% of iOS developers confirmed using XCTest/XCUITest as their primary GUI-testing platform, so Copilot already knows typical wait signatures and can auto-fill the application's bundleIdentifier [15]. When a scenario must be unified across both OSes, the model offers Appium templates: BrowserStack's latest guide explicitly names Appium the most popular mobile framework due to its uniform W3C API, enabling Copilot to reuse the same locators and methods for Android and iOS without additional branching logic.

Mobile locator peculiarities are another area where AI assistance is significant. iOS devices update significantly faster: by the end of January 2025, 68% of all iPhones were already running iOS 18 [10]. On Android, the current version 14 occupies only one-third of the market, with the remaining two-thirds spread across five major releases, as shown in Fig. 3 [11].

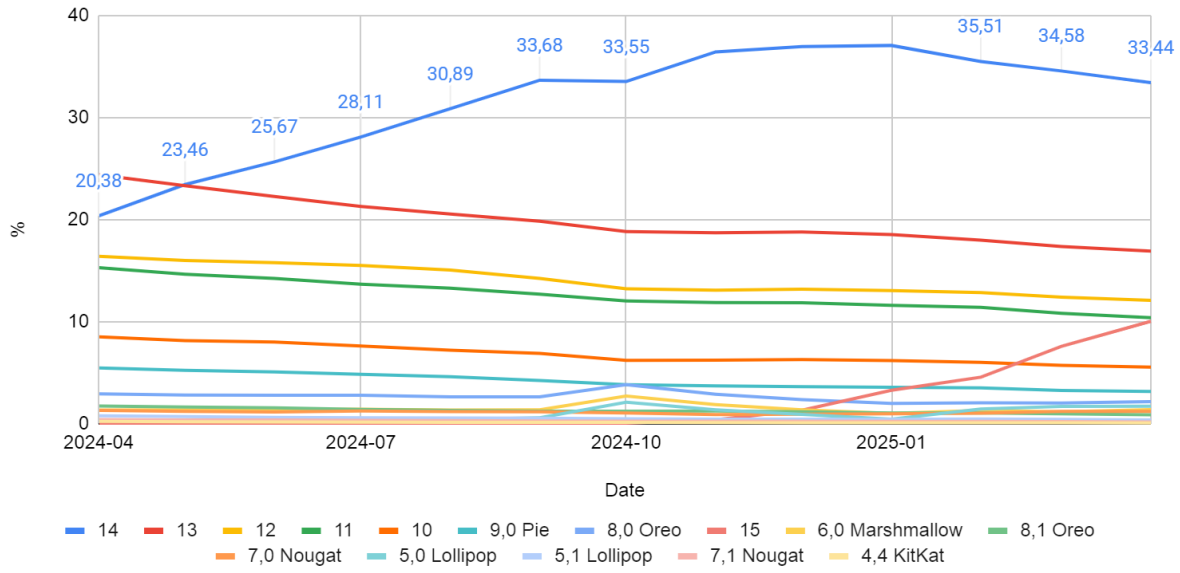

Figure 3: Android Version Market Share Worldwide [11]

Such fragmentation demands different element-finding strategies: Accessibility ID works reliably on iOS but is not always available on older Android builds; conversely, Android's resource ID remains stable but does not exist as a class on iOS. Copilot accounts for this imbalance by suggesting locator pairs and automatically arranging fallback strategies, for example, cascading searches by contentDescription and XPath.

Run stability depends not only on locators but also on proper wait management. Google's internal studies show that about 16% of tests exhibit some degree of flakiness, and 1.5% of runs end in unstable results even with unchanged code [16]. A classic antipattern is inserting a Thread.sleep; Google explicitly advises avoiding arbitrary delays and synchronizing on specific application states [17]. Copilot, trained on corpora from cloud-farm providers, often suggests wait constructs such as waitUntil { assertion } or IdlingResource, reducing the risk of intermittent failures. Since BrowserStack today provides over twenty thousand real devices for parallel

execution [18], reducing the flakiness rate immediately translates into fewer re-runs and lower costs.

Thus, the blend of IDE review, Gradle-script knowledge, and the learned Copilot model raises E2E test creation to the level where the engineer moves from simple code input to active scenario crafting and risk oversight. This is particularly clear in the mobile setting, where large Android divisions and fast iOS changes make manual test upkeep expensive. The next evolutionary step is joining generative ideas with test-coverage analysis and power-use watching systems, which we will discuss more.

Kotlin Multiplatform, thanks to its three independent source sets—commonMain, androidMain, and iosMain—allows a unified data model while compartmentalizing platform logic as needed. According to the first extensive JetBrains survey, 42.2% of developers already use KMP for shared code between iOS and Android [19], and another 22% employ Compose Multiplatform for shared UI, indicating the technology's gradual emergence as an industry standard [20]. JetBrains' testing documentation confirms that a uniform project structure simplifies the execution of standard and platform-specific tests directly from the IDE or via the Gradle command line, without additional plugins [21].

Copilot is embedded in this architecture: by analyzing build.gradle.kts and the source-set structure, it offers templates with expected and actual directives. The developer must only request to create an authentication step with a familiar interface and platform implementations. It makes a file with an expect function in commonTest and two actual implementations, importing Espresso and XCUITest. The repos for Koin and other libs show that this plan works even for DI modules, not just the UI layers, and it compiles well on all target platforms [22].

From the test engine's point of view, Copilot moves just as well between JUnit 5 and Kotest: the latter gives native KMP help and lets you run the same specification class on the JVM, iOS sim, and JavaScript engine, as shown by the three-tier testing practice discussed in a community case study [23].

A typical workflow in Android Studio begins with scenario formulation: the engineer drafts a Gherkin description or checklist directly in Copilot Chat. The model immediately generates a test class with JUnit 5 annotations, a Page Object scaffold, and correct imports; if a KMP module is present, the corresponding methods are automatically marked as expected. Manual validation of locators and refinement of non-standard steps follow, after which tests are run in parallel on multiple emulators using Android Test Orchestrator or a similar mechanism, which practitioners' experiments show reduces total execution time compared to sequential mode [24]. The completed run produces a JUnit report; with access to artifacts, Copilot can summarize failed steps and suggest locator refactoring if elements are frequently not found at a particular API level.

Despite its impressive feature set, the toolchain has limitations. The main barrier is context size: even after increasing Copilot Chat's window to 64,000 tokens, in complex monorepos, the model sometimes forgets earlier definitions and repeatedly suggests existing classes, necessitating manual engineer oversight [25]. Template-driven generation also carries the risk of propagating defects: in an empirical study of 452 Copilot-generated snippets, 24.5–32.8% contained potential vulnerabilities, underscoring the need for code review and static analysis [26]. Finally, legal concerns remain: Microsoft's policy emphasizes that Copilot may suggest fragments

subject to third-party licenses. Hence, the company recommends including secret and license-compliance checks in the CI/CD pipeline before merging into the main branch [26]. These facts indicate that even with high automation, the expert engineer is responsible for test quality and security; the AI assistant should act as an accelerator, not a replacement for engineering practices.

The current interplay of IDEs and AI helpers shows real advantages: from auto creation of Page Object and Gherkin files to adjustments for Android and iOS quirks, significantly cutting down on time spent in mundane tasks and test-infrastructure setup. Adding Copilot and similar models into IntelliJ IDEA/Android Studio lifts engineers out of the need to manually write code so they can concentrate more on test architecture and risk management; however, the stats for suggestion acceptance and found defects show that responsibility for quality stays with the expert. This impact is felt most strongly in mobile automation, where AI provides support in dealing with platform fragmentation and frequent updates from iOS. But, context restrictions plus licensing issues require scrutiny plus further controls.

4. Conclusion

In conclusion, the study demonstrates that integrating AI assistants into modern IDEs fundamentally transforms the process of developing automated tests: engineers shift from routine assembly of boilerplate code to active scenario design and risk management. Statistical data show that although Copilot and similar tools generate a substantial volume of test logic, the acceptance rate of suggestions does not exceed one-third, and only about 17–20% of automatically proposed fragments find their place in the final codebase. This indicates that the AI model already effectively offloads some routine operations; however, the ultimate quality and correctness of tests still depend on the expertise of the developer-expert.

The advantages of AI assistants are particularly pronounced in mobile automation, where Android fragmentation and the rapid iOS update cycle create significant technical barriers. Tools capable of analyzing Gradle scripts and the Kotlin Multiplatform architecture automatically generate Page Object scaffolds, insert locators with fallback strategies, and produce cross-platform steps, thereby reducing the time required for initial setup and infrastructure maintenance by several times. Simultaneously, Copilot’s great flexibility with internal repositories and the style of code for projects helps in generating tests that are closer to what the team wants. However, there are limits on the size of the context window and risks that vulnerabilities could be copied, which means developers need to keep watching plus adding more tools to help check quality. Increasing the chat-suggestion window to 64K tokens only partly helps with earlier definitions being forgotten; studies show up to one-third of generated code snippets may have defects or be subject to third-party licenses. Therefore, static analysis, test-coverage validation, and license-compliance scans should be integrated into a CI/CD pipeline.

The next step in improving AI helpers would be to tightly integrate them with analytics tools: automatic check of test coverage, monitoring for problems, and power-use study during real-device tests. Combining offered ideas with quality measures will speed up the automated test-making process and improve trustworthiness and effectiveness at every stage of the application life cycle.

Thus, contemporary AI models joined with IDE capabilities lift test automation to a new level, freeing engineers from routine work and enabling them to pay more attention to architecture and testing strategies. But in the end, humans shoulder responsibility for quality: the AI assistant remains an accelerator that requires careful, judicious use and added oversight.

References

[1] "Technology | 2024 Developer Survey," *Stackoverflow*. https://survey.stackoverflow.co/2024/technology#worked-with-vs-want-to-work-with (accessed Apr. 22, 2025).

[2] Y. Gao, "Research: Quantifying GitHub Copilot's impact in the enterprise with Accenture," *The GitHub Blog*, May 13, 2024. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/ (accessed Apr. 23, 2025).

[3] V. Vasudevan, "Github Copilot Adoption Trends: Insights from Real Data," *Opsera*, Apr. 29, 2025. https://www.opsera.io/blog/github-copilot-adoption-trends-insights-from-real-data (accessed Apr. 30, 2025).

[4] Jash Unadkat, "Difference between Mobile and Web Application Testing," *Browser Stack*, Dec. 13, 2024. https://www.browserstack.com/guide/differences-between-mobile-application-testing-and-web-application-testing (accessed Apr. 24, 2025).

[5] J. Rossignol, "Apple Reveals How Many iPhones Are Running iOS 18," *MacRumors*, Jan. 29, 2025. https://www.macrumors.com/2025/01/29/ios-18-adoption-stats/ (accessed Apr. 25, 2025).

[6] M. Ulianov, "How GitHub Copilot Promotes Junior Developers to Mid-Level," *FullStack Labs*, Feb. 10, 2025. https://www.fullstack.com/labs/resources/blog/how-github-copilot-promotes-junior-developers-to-mid-level (accessed Apr. 26, 2025).

[7] "State of Developer Ecosystem Report 2024," *JetBrains*, 2024. https://www.jetbrains.com/lp/devecosystem-2024/ (accessed Apr. 26, 2025).

[8] "The State of Developer Ecosystem in 2021," *JetBrains*. https://www.jetbrains.com/lp/devecosystem-2021/kotlin/ (accessed Apr. 27, 2025).

[9] "Full compatibility and enhanced authentication for GitHub Copilot in JetBrains IDEs 2024.3 - GitHub Changelog," *The GitHub Blog*, Dec. 04, 2024. https://github.blog/changelog/2024-12-04-full-compatibility-and-enhanced-authentication-for-github-copilot-in-jetbrains-ides-2024-3/ (accessed Apr. 28, 2025).

[10] J. Clover, "iOS 18 Installed on 76% of iPhones Introduced in the Last Four Years," *MacRumors*, Jan.

24, 2025. https://www.macrumors.com/2025/01/24/ios-18-adoption-rate/ (accessed May 01, 2025).

[11] "Android Version Market Share Worldwide," *StatCounter Global Stats*, Apr. 2025. https://gs.statcounter.com/android-version-market-share (accessed May 03, 2025).

[12] "AndroidX Test Library," *Mvn repository*. https://mvnrepository.com/artifact/androidx.test.espresso/espresso-core (accessed May 07, 2025).

[13] "UI automator," *Mvn repository*. https://mvnrepository.com/artifact/androidx.test.uiautomator/uiautomator (accessed May 09, 2025).

[14] L. Cruz and R. Abreu, "On the Energy Footprint of Mobile Testing Frameworks," *IEEE Transactions on Software Engineering (TSE)*, Oct. 2019, doi: https://doi.org/10.48550/arxiv.1910.08768.

[15] A. Crudu, "Unit Testing vs UI Testing in iOS Development - When to Use Each for Optimal Results," *MoldStud*, Apr. 06, 2025. https://moldstud.com/articles/p-unit-testing-vs-ui-testing-in-ios-development-when-to-use-each-for-optimal-results (accessed May 10, 2025).

[16] J. Micco, "Flaky Tests at Google and How We Mitigate Them," *Google Testing Blog*, May 27, 2016. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html (accessed May 11, 2025).

[17] "How Much Testing is Enough?" *Google Testing Blog*, Jun. 15, 2021. https://testing.googleblog.com/2021/ (accessed May 12, 2025).

[18] "Pricing," *Browser Stack*. https://www.browserstack.com/pricing?cycle=annual (accessed May 13, 2025).

[19] A. Anisimov, "Results of the First Kotlin Multiplatform Survey," *JetBrains*, Apr. 28, 2021. https://blog.jetbrains.com/kotlin/2021/01/results-of-the-first-kotlin-multiplatform-survey/ (accessed May 15, 2025).

[20] E. López-Mañas, "The State of Developer Ecosystem in 2023," *JetBrains*. https://www.jetbrains.com/lp/devecosystem-2023/kotlin/ (accessed May 16, 2025).

[21] "Test your multiplatform app – tutorial," *JetBrains*, May 19, 2025. https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-run-tests.html (accessed May 20, 2025).

[22] "Multiplatform expect / actual module support," *GitHub*, Dec. 07, 2024. https://github.com/InsertKoinIO/koin-annotations/issues/105 (accessed May 17, 2025).

[23] "Kotlin Multiplatform's three levels of testing with Kotest," *Dev Community*, Jan. 21, 2023.

https://dev.to/bjornvdlaan/kotlin-multiplatforms-three-levels-of-testing-with-kotest-3e24 (accessed May 18, 2025).

[24] R. Lukasik, "Running our Android Espresso tests in Parallel," *Medium*, Oct. 27, 2023. https://rlukasik.medium.com/running-our-android-espresso-tests-in-parallel-2c13ef9231f7 (accessed May 19, 2025).

[25] "Copilot Chat now has a 64k context window with OpenAI GPT-4o - GitHub Changelog," *The GitHub Blog*, Dec. 06, 2025. https://github.blog/changelog/2024-12-06-copilot-chat-now-has-a-64k-context-window-with-openai-gpt-4o/ (accessed May 19, 2025).

[26] Y. Fu, "Security Weaknesses of Copilot Generated Code in GitHub," Arxiv.org, 2023. https://arxiv.org/html/2310.02059v2 (accessed May 28, 2025).