

## BEST PRACTICES FOR SYSTEMATIZING ROUTINE PROCESSES IN MOBILE DEVELOPMENT PROJECTS

*Troshchyi D., Kuznetsova Yu.*

*In mobile development, a significant amount of time is spent on repetitive tasks: setting up the environment, configuring the build, checking pull requests, updating dependencies, running tests, etc. Despite the availability of automation tools such as CI/CD, templates, and scripts, their implementation is often complicated by technical, organizational, and mental barriers. Typical difficulties include unstable environments, fragmented solutions, lack of support, and the absence of unified approaches. Automation is often implemented as a one-off initiative without a long-term vision, which reduces its viability. An analysis of recent studies shows that most of them focus on testing, without covering other critical stages of the application lifecycle. Existing research rarely takes into account the specifics of mobile platforms, such as the presence of markets, signatures, and SDK updates. As a result, teams create their own solutions that are difficult to scale or quickly become obsolete. The article summarizes the problems that developers face at all stages, from project configuration to maintenance. It has been established that routine tasks affect not only the pace of work and quality, but also team motivation, contributing to burnout and the accumulation of technical debt. Therefore, we propose creating a framework that systematically links typical tasks with tools, practices, and viability criteria. This approach will reduce fragmentation and increase the stability and efficiency of mobile development.*

### Introduction

In mobile app development, a significant part of daily work consists of repetitive tasks that may seem minor at first glance. These include environment and build configuration [1], request pool checking [2], building, dependency updates [3], test running [4], and so on.

Such routine tasks accompany virtually every stage of an application's life cycle, from design to maintenance, and they often take up a significant portion of a developer's working time.

The next step after identifying such tasks is to try to improve their efficiency through automation [5], templating [6], or organizational changes. However, in practice, even the most obvious and simple improvements are often hindered by difficulties such as lack of time, technical limitations, complexity of maintenance, or simply the absence of unified approaches within the team. As a result, some repetitive tasks are not solved systematically, but are done manually each time, despite the availability of tools or experience from previous projects.

The question is not only how to perform a routine task efficiently, but also why attempts to solve it often encounter resistance: technical, organizational, or mental.

In mobile development practice, such bottlenecks only deepen over time, especially in the context of accelerated release cycles, constant platform updates, and an increasing number of dependencies.

## 1. Complexity of solving routine tasks

At first glance, most repetitive tasks in mobile development seem pretty basic: set up a build, check a variable in CI (Continuous Integration), handle an API (Application Programming Interface) error, write a basic unit test. These are things that don't seem to need any fancy solutions. However, this is precisely where the problem lies – attempts to automate or standardize them often encounter hidden barriers that, when combined, create a significant obstacle.

*First, there are technical limitations.* Even modern platforms – Android and iOS – have their own peculiarities that complicate the unification of approaches. For example, updating libraries in an Android project may seem simple, but in a project with a large number of modules and no centralized location for dependencies, this turns into a series of manual checks and edits in several configuration files. In projects with complex CI/CD pipelines, adding new steps, even formally simple ones, sometimes requires diving into Jenkins or Bitrise documentation, and sometimes simply writing a separate Bash script.

*Second, organizational barriers.* In many teams, attempts to systematize or automate routines are not supported by management. The reasons are trivial: tight deadlines, a focus on visual functionality rather than processes, or a lack of internal culture of fixing what works. In such conditions, even obvious improvements, such as introducing a template generator for creating new screens, are put off until later, which never comes.

*Third, the human factor.* Many developers find it easier to do a task themselves than to spend time on «automation for automation's sake». This is partly justified; indeed, not every routine task requires a technical solution. But when there are dozens of such actions every day, their total weight becomes significant. This is especially true if these tasks are performed by different people with varying levels of quality, which creates problems in terms of support, documentation, and knowledge transfer.

*Another common problem is technical debt associated with previous attempts at a solution.* For example, a team may have implemented a tool to automate assembly, but over time it stopped working due to API changes or platform limitations, and no one took on the task of maintaining it. As a result, the tool

remained in the project but is not used. New developers are forced to look for workarounds, which are again not automated.

*Equally important is that mobile development is often ignored in general automation practices.* Many of the existing solutions and recommendations were developed in the context of web or backend development, where the infrastructure is more predictable: deployment happens without involving app stores, new changes can be uploaded to the server at any time, and CI/CD is often already integrated into the platform. In mobile development, the situation is different – SDK (Software Development Kit) updates, strict App Store and Google Play requirements, release signing – all of this create an additional workload that is difficult to automate without deep knowledge of the platform.

So, the problem of solving «routine» tasks in mobile development is not just the lack of technical tools. Often, the solution already exists, but it doesn't take root because it requires regular support, team agreement, and flexibility in changing processes. In addition, there is a lack of a common approach to automation and efficiency improvement: teams create their own templates, scripts, and utilities, but these solutions remain local, difficult to scale, and transferable to other projects. As a result, even with the technical capabilities in place, a significant portion of repetitive tasks continue to be performed manually, with all the associated time losses.

## **2. Overview of typical difficulties in research**

Despite the widespread popularity of mobile development in practice, the scientific base devoted to systematic analysis and routine problem solving in this context remains rather limited. Most publications either focus on individual aspects of the mobile application lifecycle, such as testing, or are of a general nature that is not adapted to the specifics of mobile engineering. At the same time, there are virtually no publications that comprehensively cover all phases of development, from project configuration and requirements engineering to release stages, monitoring, and feedback processing.

In their work «A survey on the practices of mobile application testing» [7], the authors conduct a large-scale survey of mobile developers on their approaches to testing. The study confirms that manual testing prevails, while automated tools are implemented fragmentarily, often without proper integration into the overall development infrastructure. The low level of unit test coverage, lack of UI gesture support, and disregard for scenarios with contextual interaction

(geolocation, camera, push notifications) are highlighted separately, turning many checks into repetitive routines.

A broader overview is provided in «A Survey on Mobile App Development Approaches with the Industry Perspective» [8], which examines not only technical approaches but also organizational development models. The authors point out that mobile engineering is often based on Agile or MVP cycles, but there are almost no established models that adapt classic software engineering principles to the specifics of mobile development. In particular, it is noted that handling changes in requirements, adapting to platform updates, and rapid prototyping remain predominantly manual processes, with automation affecting only a small portion of the life cycle, confirming the key thesis about the limitations of solutions for «routine» tasks.

The paper «The Applicability of Automated Testing Frameworks for Mobile Application Testing» [9] reviews 56 scientific papers analyzing the capabilities of existing frameworks for mobile testing. The study is systematic and covers the evolution of methods from keyword-driven to hybrid and AI-assisted approaches. However, even the most modern tools demonstrate limited adaptation to mobile conditions: poor support for complex scenarios, high configuration requirements, and a lack of flexibility to support different platform versions. These factors significantly limit the effective resolution of routine tasks during the testing phase, especially when it comes to checking UI components and regressions.

The study «How do Developers Test Android Applications» [10] confirms that even in Android projects with stable teams, automation is used to a limited extent. The authors emphasize that developers often consider testing to be either secondary or not worth the cost. This creates a culture of manual functionality review, where stability is ensured not by tools, but by the personal time and effort of developers. Ultimately, this model leads to repetitive actions that remain in the project for years.

Overall, the sources analyzed show that even where there are attempts to solve routine tasks, such as testing, these solutions are highly specialized, often not scalable, and require significant human resources to maintain. The most systematic of the existing approaches focus exclusively on testing but hardly touch on other critically important stages of the life cycle. The lack of cross-stage integration and methodological consistency confirms the need for a framework that will combine routine tasks with the appropriate tools and practices within a single structure adapted to mobile development.

### **3. Problems of repetitive actions at different stages of development**

Mobile app development consists of repetitive actions that occur at every stage: from the first launch of the IDE (Integrated Development Environment) to technical support after release. Although at first glance each of these tasks seems fairly simple on its own, together they create a significant workload that is not always easy to reduce, even with the right tools. The reason is not only technical limitations, but also the fact that most solutions are created on an ad hoc basis, without a systematic approach, long-term vision, or responsibility for support.

Problems begin as early as the configuration stage of a new project. Even within a single platform (e.g., Android), the structure of an application can vary from team to team. Some use modular architecture, while others use a monolithic one. The same applies to build schemes, signing config, integration with CI/CD (Continuous Integration / Continuous Delivery / Deployment), Firebase, analytics, libraries for working with HTTP (HyperText Transfer Protocol), DI (Dependency Injection), and so on. As a result, instead of reusing typical configurations, developers are forced to reconfigure everything from scratch each time, copying fragments from previous projects or documentation. In the best case, internal boilerplate is used; in the worst case, each engineer creates a structure based on their own experience.

Any attempt to standardize these actions often runs into a lack of time, support, or unwillingness to change what works. Writing the code itself also involves repetitive patterns and complexities that are difficult to automate.

In addition to generating basic elements (ViewModel, layout, models), there is a lot of manual work involved in creating the UI. If a ready-made component library is not used, most of the interface is laid out manually, either from scratch or based on Figma layouts. Moreover, adapting to different screen sizes (especially for Android) is still a challenging task. Automatic UI tests do not always guarantee correct rendering, and visual snapshot tests, although popular, often require manual approval and are too sensitive to minor changes.

Another common problem is code duplication. Its appearance is a direct consequence of the lack of centralized solutions. For example, if the team does not have shared components for navigation, error handling, logging, or status display (e.g., loading/error/empty), each screen is implemented in its own way. Over time, dozens of classes with similar logic appear, which are difficult to update simultaneously or extract into a common layer. Ways to reduce duplication, such as templates or wrappers, are often not implemented due to lack of time, and repetitive code accumulates.

The process of writing code itself is also part of the «routine» that people try to speed up. For example, with autocomplete, snippets, internal plugins, or generators. But supporting such solutions in complex projects requires constant adaptation to changes. Often, a team starts out enthusiastically, creating a set of utilities or templates, but after a few months, they stop working with new versions of tools or don't support changed requirements. Without a defined update process, these developments are simply forgotten, and routine work returns.

CI/CD is an area that is formally best suited for automation, but in practice it is also vulnerable. Any update to SDK or Google Play requirements can cause failures. The complexity lies in the fact that the build process is a combination of scripts, keys, flavors, signatures, environment variables, and saved artifacts.

Often, CI/CD configuration depends on the person who created it, and if that person leaves the team, adapting or scaling pipelines becomes a problem. In such cases, instead of automation, the project reverts to manual build delivery because it is easier in the short term. Working with tests is no exception.

Ideally, tests should be part of the development process, but when the application structure changes, old tests start to break. If there is no clear responsibility for maintaining the test environment, or if the tests themselves were created without following best practices, then any large-scale update will cause them to fail en masse, and they will simply be disabled. This also applies to UI tests, which often depend on the stability of screens, locales, backend connections, animation timings, and so on.

The release and regression stages remain a serious source of routine work. Preparation for publication should formally be as automated as possible, but in reality it is often accompanied by manual verification, copying change descriptions, and preparing screenshots and specifications for different app stores. Even minor actions, such as updating a version, publishing a release branch, or generating a changelog, are performed manually in many projects, which wastes time and creates risks of errors. The reason is that automation is not fully implemented or is outdated. As a result, when the release deadline approaches, the team simply repeats the same actions they have already performed dozens of times, without any changes.

A separate challenge is post-release regression, when new functionality unexpectedly starts to break. This often happens not because something is fundamentally broken, but because the change was not sufficiently tested under normal conditions. Testing may have been only local, or performed on a different build flavor, or someone manually fixed something in production during the build. In this case, the search for causes begins: is it a bug in the code, in the configuration, in CI, in Google Play, or in the tester's environment? This is a classic situation

where the team spends a day or two localizing a bug that arose due to an obscure deviation in the settings or library version.

Even worse is when complaints from users appear after release, and the team only learns about the problem from comments in the market. Gathering information about a bug in such a situation is a manual process: checking logs, checking versions, manually reproducing the bug, reconfiguring the environment. If there is no clear habit of keeping a list of release changes, logging what could have broken, or at least performing regular smoke tests, the process turns into a repetition of the same steps: check on an emulator (or simulator), run on an old device, update the SDK, clear the cache.

In the support phase, the routine does not disappear – it changes form. Tracking crashes, analyzing feedback, updating dependencies, preparing hotfixes – all of this takes time. Often, the team does not have a clear structure for processing incoming information: someone reads reviews in the market, someone looks at Crashlytics, and someone learns about a bug from a colleague on Slack. Systematizing this work is considered an indirect task, and there are not enough resources for it.

All these problems have a common source: decisions to reduce routine work are made on a case-by-case basis rather than as part of a systematic vision. Even if a single task is automated or simplified, this does not guarantee that the next one will be solved in the same way. Without a coordinated approach, an update process, documentation, and internal accountability, solutions remain vulnerable. And even the best ones are forgotten over time, giving way to the next iteration of manual work, which is faster because it is familiar.

#### **4. Impact of repetitive tasks on development efficiency and quality**

«Routine» in development is not always perceived as a problem – it is often disguised as familiar actions that simply need to be done. But this is precisely where its insidiousness lies: these actions do not cause direct resistance, are not considered mistakes, are not recorded in Jira as bugs, and yet they consume time, energy, and focus every day.

As a result, not only speed suffers, but also the quality of development and, just as important, the state of the team itself.

First of all, routine directly affects productivity. When a developer spends an hour every day repeating the same actions – building, updating dependencies, copying templates, manual testing, checking configurations – that hour is lost for tasks that have strategic or creative value. These may seem like minor issues,

but over the course of a week, they add up to half a day, and over the course of a month, several full working days. In projects without well-coordinated automation, such losses become a background load that everyone ignores – but which constantly slows down the overall pace of work.

These repetitive actions also reduce product quality. When a team is working under deadline pressure, any complication, such as manually creating a build or checking settings, forces them to sacrifice other things: test coverage, refactoring, or even basic change verification. If the build fails the first time, the release is often postponed or released with temporary fixes that are then forgotten to be removed. This approach creates technical debt that does not accumulate dramatically but eats away at the code base from within.

At the team level, routine leads to uneven workloads. Often, there are one or two people who know how to assemble a release, configure CI, or quickly check staging. Everyone else waits until they are free from other tasks. The result is dependence on specific people, bottlenecks in the process, and a loss of flexibility. If such an engineer takes a vacation or leaves, the team loses a whole area of knowledge that has to be rebuilt from scratch.

Another important consequence is burnout. Constantly performing monotonous tasks that do not involve any challenge or intellectual effort reduces engagement. A developer who, instead of solving an interesting problem, spends an hour setting up the environment or manually compiling a build, quickly loses motivation. This is especially noticeable in startups or small teams where the number of people is limited, and the tasks are varied. Where there is no system, routine falls on the shoulders of the most active members, and at some point, they simply burn out.

Equally important, routine creates a false impression of efficiency. When a process is built on dozens of manual actions, the team feels that everything is under control because everyone knows what to do and when to do it. But in reality, control becomes illusory: all it takes is a new app store policy or a change in tools, and the system falls apart because nothing is documented, nothing is standardized, and every action is only in the head of a specific team member.

Ultimately, even with good intentions, the lack of a systematic approach means that any implementation – whether CI, code templates, or a build tool – remains fragmented. Without a shared vision and support, such initiatives disappear along with those who initiated them. And even if the next team decides to do things right, it will repeat the same cycle: manual work, fragmented improvements, oblivion, manual work.

## Conclusions

An analysis of the daily practices of mobile developers has shown that routine tasks not only remain present in modern workflows, but also have a significant impact on development speed, product quality, and team collaboration. Despite the availability of technical solutions such as automation, templates, and CI/CD tools, most attempts to reduce repetitive tasks remain isolated. They arise as a response to local difficulties, are not integrated into the overall context of the development lifecycle and are not supported by long-term practices.

The problem is further complicated by the fact that much of the routine is perceived as something normal, something that is supposed to be that way. In reality, however, this is one of the hidden barriers to scaling, effective onboarding of new team members, and ultimately product stability. The lack of common standards, transparent processes, and responsibility for maintaining even the simplest automations leads to their gradual decline and, consequently, a return to manual execution.

This study did not aim to provide a comprehensive solution, but rather to identify the problem area, describe it, and structure it. The result was the formation of a comprehensive vision of which stages of mobile development are accompanied by repetitive tasks, how these tasks manifest themselves in a real environment, why the implemented solutions often do not work, and what consequences this has for the team.

Further research should focus on building a practical framework that would allow typical routine tasks to be matched with specific tools, approaches, or organizational practices that have proven effective in a similar context. Such a framework should not only catalog existing solutions but also create a convenient structure for their evaluation, adaptation, and scaling. The focus should be not only on automation, but also on the viability of solutions: how easy they are to implement, maintain, transfer within a team, and apply in different conditions. Over time, such a model could become the basis for unifying approaches to routine tasks in mobile development, which would significantly increase the predictability, stability, and productivity of teamwork.

## References

1. Build your app from the command line. Android Developers. URL: <https://developer.android.com/build>

2. About pull requests. GitHub Docs. URL: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests>
3. Add build dependencies. Android Developers. URL: <https://developer.android.com/build/dependencies>
4. Fundamentals of testing. Android Developers. URL: <https://developer.android.com/training/testing/fundamentals>
5. What is Software Automation? GeeksForGeeks. URL: <https://www.geeksforgeeks.org/what-is-software-automation>
6. Create app templates. Android Developers. URL: <https://developer.android.com/studio/projects/templates>
7. Santos I., Filho J., Souza S. A survey on the practices of mobile application testing. Proceedings of the XLVI Latin American Computing Conference. 2020. P. 232–241. URL: <https://doi.org/10.1109/CLEI52000.2020.00034>
8. Patidar A., Suman U. A Survey on Mobile App Development Approaches with the Industry Perspective. International Journal of Open Source Software and Processes. 2022. Vol. 13(1). P. 1–17. URL: <https://doi.org/10.4018/IJOSSP.300754>
9. Berihun N., Dongmo C., Van der Poll J. The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review. Computers. 2023. Vol. 12(5). P. 97. URL: <https://doi.org/10.3390/computers12050097>
10. Linares-Vásquez M., Bernal-Cárdenas C., Moran K., Poshyvanyk D. How do Developers Test Android Applications?. Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2017. P. 613–622. URL: <https://doi.org/10.1109/ICSME.2017.47>