Mobile Architecture Task Force

Why we think Flutter will help us scale mobile development at Nubank

A detailed report of the criteria and study we conducted to decide to use Flutter as our main technology for cross-platform mobile development.

nubank

October 2019

Thank you for reading

Authors

Alexandre Freire, André Moreira, Rafael Ferreira, Rodrigo Lessinger, Victor Maraccini and Vinícius Andrade.

Thanks to advisors

Bruno Tavares, Caio Gama (on usability testing methodology), Edward Wible, Fellipe Chagas, Francesco Garcia, Guilherme Neumann (on design), Hugh Strange (on product), Igor Borges, Luiz Dubas, Max Miorim, Rafael Ring, Rodolfo Fiuza, Thiago Moura and Wilker Lúcio.

Thanks to testers

Alexandre Freire, Ana Luisa Bavati, Ana Valeije, Daniel de Jesus Oliveira, Eden Ferreira, Edward Wible, Fuad Saud, Joel Junio, Lucas Ferreira, Lucas Mafra, Luiz França, and Rafael Maruta.

Abstract

Nubank is a high growing fintech with a fast development pace. We're cloud-first, mobile-first and consider ourselves a technology company. Having to support two platforms for our mobile applications brings challenges to the development organization; access to very specialized talent pools where market competition is fierce; and a high entry barrier for mobile development.

Looking to address some of these problems we've decided to investigate if a cross-platform solution could attend to Nubank business goals and deliver high-quality apps.

This study presents the result of a Research & Development project conducted by a small team at Nubank tasked with evaluating three cross-platform technologies: Flutter, Kotlin Native, React Native; in eleven criteria evolved over time by the team that represents what's important for Nubank in its current context.

We will explain how each criterion was graded in a [0-5] scale, and show why we decided to adopt Flutter as the main technology for cross-platform mobile development at Nubank.

History

Nubank's mobile apps started in 2014 as two separate native projects, written in Objective-C on iOS and Java on Android. Swift and Kotlin were released and adopted as primary development languages for the mobile platforms at Nubank around 2016. By mid-2017, the NuConta project started as a new React Native application, investing in a cross-platform technology to increase development speed and allow for smaller autonomous teams. This resulted in three fundamentally different ways of writing mobile apps: native iOS, native Android and cross-platform.

We had separate teams using different sets of tools, languages and conventions. We found it was hard to share knowledge and technologies with each other. In addition, there was a high barrier of entry for new developers to join mobile development, caused by either having to learn different languages or by high setup and maintenance costs of the underlying platforms.

We encourage teams to move autonomously at Nubank, so we were happy to see different teams trying different things, but at the same time the cost described above was increasing. As we let teams diverge with many competing ideas, there comes a time where we decide to staff horizontal engineering support teams to study all these localized efforts, pick the best and help all of engineering standardized.

The Mobile Architecture Taskforce was assembled in early 2019 to try and find a single solution for developing cross-platform mobile applications at Nubank, taking the company's requirements and culture into account to make an informed and balanced decision on the technology to be used for mobile development in our future.

The task force's vision was:

> *"Regardless of the specialization of its members, squads will be autonomous and productive to develop the mobile application on a single architecture and set of conventions, using the same programming language to deliver value continuously to delight customers."*

And it's Mission:

Working For: Product-facing squads who need to develop features on the mobile app, by the end of February, this team will:

1. Gather data
2. Study alternatives
3. Document
4. Foster participation and feedback from the Engineering chapter
5. Decide

... between the considered cross-platform development alternatives (Kotlin Native, React Native and Flutter).

Unlike other local efforts in the past, this team will have 3 dedicated people to help converge on a solution with buy-in from all of engineering.

Metho-dology

We first iterated to align on which evaluation criteria were important for Nubank. Initially we were worried about technical issues and limitations of the frameworks, so thought about some non-functional requirements like:

- **Startup time:** both cold and hot startup of the platform, not considering userland code;
- **FPS:** does the default path outputs code that runs on 60 Frames per Second;

As we discussed these with our advisors we realized that the people that were going to be impacted by the decision were the most important criteria. Having different technologies and frameworks was generating a lot of anxiety, especially among the native mobile specialists we had in the team.

Uncertainty around what would happen if React Native became a standard caused anxiety: would I have to go back and learn mobile all over again? Would I be considered more “junior” in this new stack? What if I was really proud of my native specialty and my ability to solve hard problems in that space?

So we improved our list of criteria and focused on the ones that reflected the impact we wanted to bring to our team.

We then set out to gather evidence and agree on a subjective score, from very low to very high, for each of them by using different techniques like:

- testing a Flutter version of one of our features in production
- analyzing communities, repositories, and resources available for each platform
- engaging in conversations with specialists, teams, and companies behind the development of the platforms
- implementing a clone of one of our features as a stand-alone app in the 3 different platforms
- conducting an internal usability test, were a novice and senior engineers made changes to the feature in apps described above
- gathering internal data through surveys focused on mobile developers and designers
- conducting presentations, debates and team visits to discuss our findings, hear engineers and senior advisors’ opinions, incorporate their feedback and answer their questions.

Methodology

As we got data and answers we gave ourselves a deadline to decide, we found this to be crucial for us to not prolong the taskforce and getting diminished returns in the quest for a perfect answer. We had contention in the team up to the last minute and a technique that really helped was running a “catastrophic scenario” role play. Imagine if everything that can go terribly wrong with this decision does. How would it play out for each option? We discussed scenarios like “Google decides to kill Flutter” and “Facebook abandons React Native” and encouraged the team to bring out their worst fears and then play out how Nubank would react in “what if... ?” scenarios.

Finally, we made sure to invest a lot of conscious effort into communication. We held regular status update meetings with the Engineering Management team, and kept tabs on engineers who were engaging with us either on Slack, in our live presentations and debates, or in our visits to teams.

When we announced the decision to the engineering chapter nobody was surprised and everyone felt they had the space to participate and share their opinions and concerns.

OTTO NASCARELLA
Software Engineer

“Our team was invited to a presentation were we learned about all options in the choice. What called my attention was what came next: they present the criteria that was chosen to make the decision and the rationale behind each of them. After that, they also presented some of the experiments that would be performed to make a decision relative to each criteria. The care taken to present all these attributes, and the total openness so that we could give them input into each of them, was fundamental so that the whole team felt as an indispensable part of the process, and for all of us to understand the seriousness of the work that had been done: this made us feel completely safe with whatever the decision might come out of the taskforce’s work.”

Testimony of senior engineer Otto Nascarella on his team’s engagement with the task force.

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

01 Development experience

DEVELOPER EXPERIENCE

Factors that contribute to enable a developer to deliver and to be productive on the mobile app.

Examples:

- Hot reload
- Component visibility
- Debugger tooling
- IDE integration
- Test Tooling

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

02 Long-term viability

LONG-TERM VIABILITY

Confidence that the platform maintainer will keep supporting it in the long-term (five years) and the likelihood that the community will be able to support the project if the maintainer decides not to continue.

Examples:

- Adoption by big companies
- Size of the community (number of core contributors, outside contributors, etc...)

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

03 No platform specialization

NO PLATFORM SPECIALIZATION

An engineer should be able to write mobile code for the product without differentiating between Android and iOS. The code should look and behave the same on Android and iOS, with low occurrence of OS-specific crashes/problems.

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

04 Incremental abstraction cost

INCREMENTAL ABSTRACTION COST

The cost of extending the NuDS platform for each product task and the friction of centralizing the work on extensions, if required.

Examples:

- Does adding a new component require a dependency on a horizontal squad to move forward?
- How hard is it to add a new component? How about making it accessible/performant…?

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

05 Non-linear abstraction risk

NON-LINEAR ABSTRACTION RISK

Risk of sudden requirement of large, disproportionate rewrites of our internal abstraction.

Examples:

- Adding an extra lifecycle method requires refactoring most of the components
- A single new NuDS component that requires non-trivial changes across the entire codebase

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

06 Learning resources

LEARNING RESOURCES

Amount and quality of available external learning resources, such as the official documentation, StackOverflow answers, books, conferences and courses.

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

API/TOOLING STABILITY

Platform API or tooling changes that require changing our internal code.

Examples:

- Base API changes or dependency changes that make it backwards-incompatible
- Changes to native (OS) components that break internal cross-platform behavior

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

08 App Store Restrictions

APP STORE RESTRICTIONS

Risk of Apple or Google restricting our app in any way because of our underlying platform.

Examples:

- Flutter UX not matching Apple's HIG (Human Interface Guidelines)
- The possibility of Over The Air updates on React Native/Flutter becoming a blocker for Apple

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

09 Capabilities limitations

CAPABILITY LIMITATIONS

Having all the latest OS/device features accessible to us in the chosen platform.

Examples:

- Android Minimized apps
- Apple Watch support

Evaluation Criteria

The eleven criteria were listed and prioritize:

PRIORITY

10 Roadmap

ROADMAP

Visibility and long-term roadmap diverging from Nubank's interests.

Evaluation Criteria

The eleven criteria were listed and prioritize:

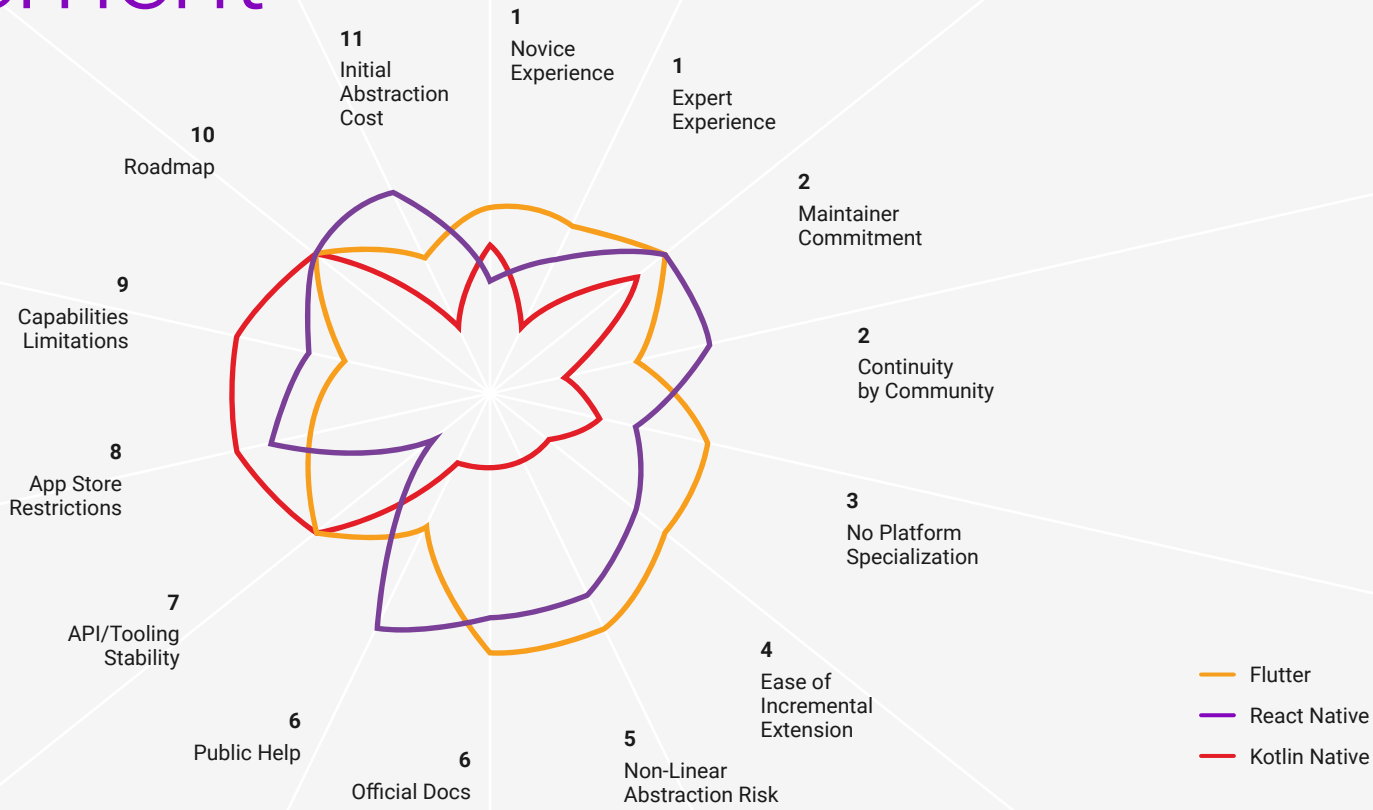
PRIORITY

11 Initial abstraction cost

INITIAL ABSTRACTION COST

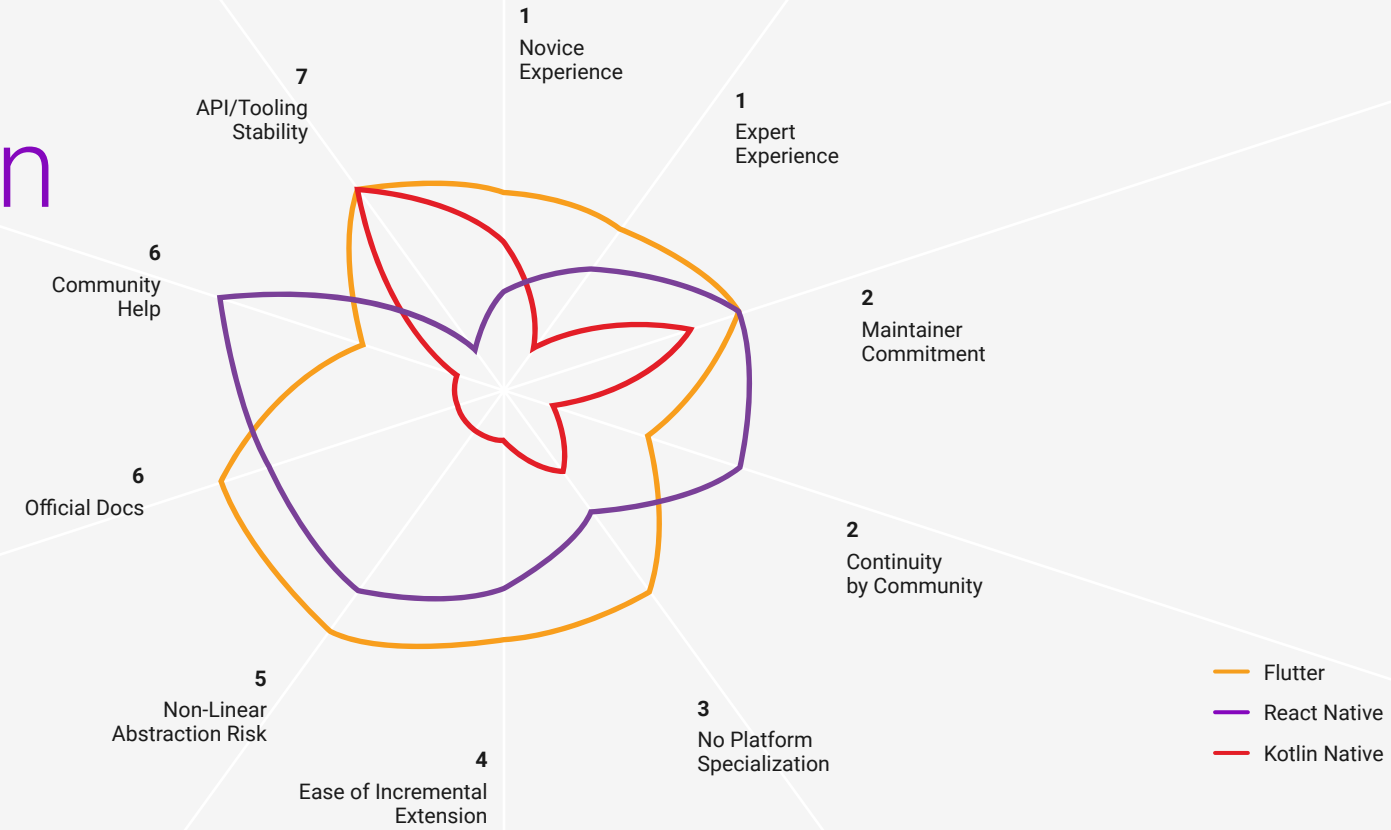
One-time cost of providing a minimal abstraction for product teams to work.

Measurement results

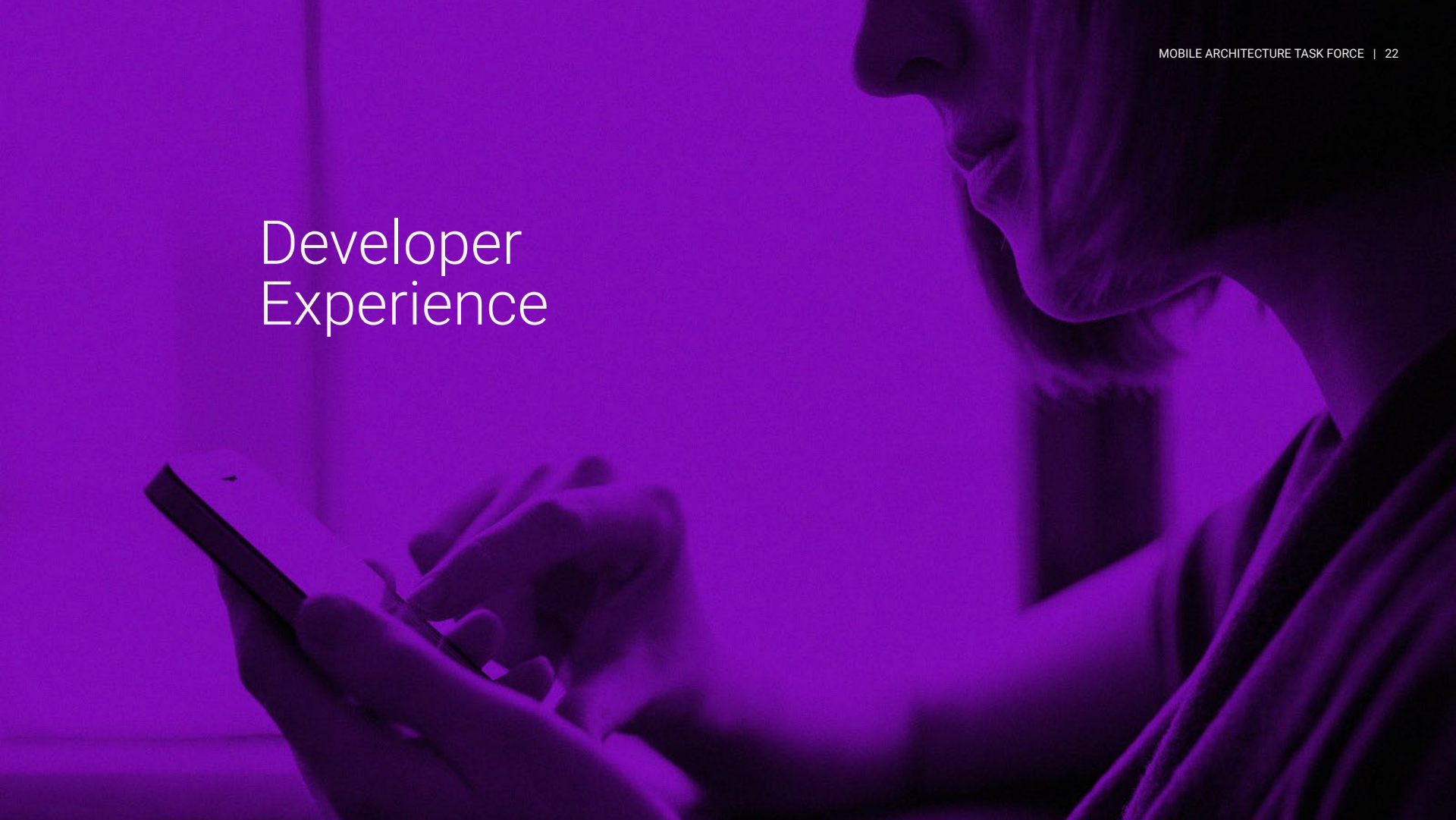


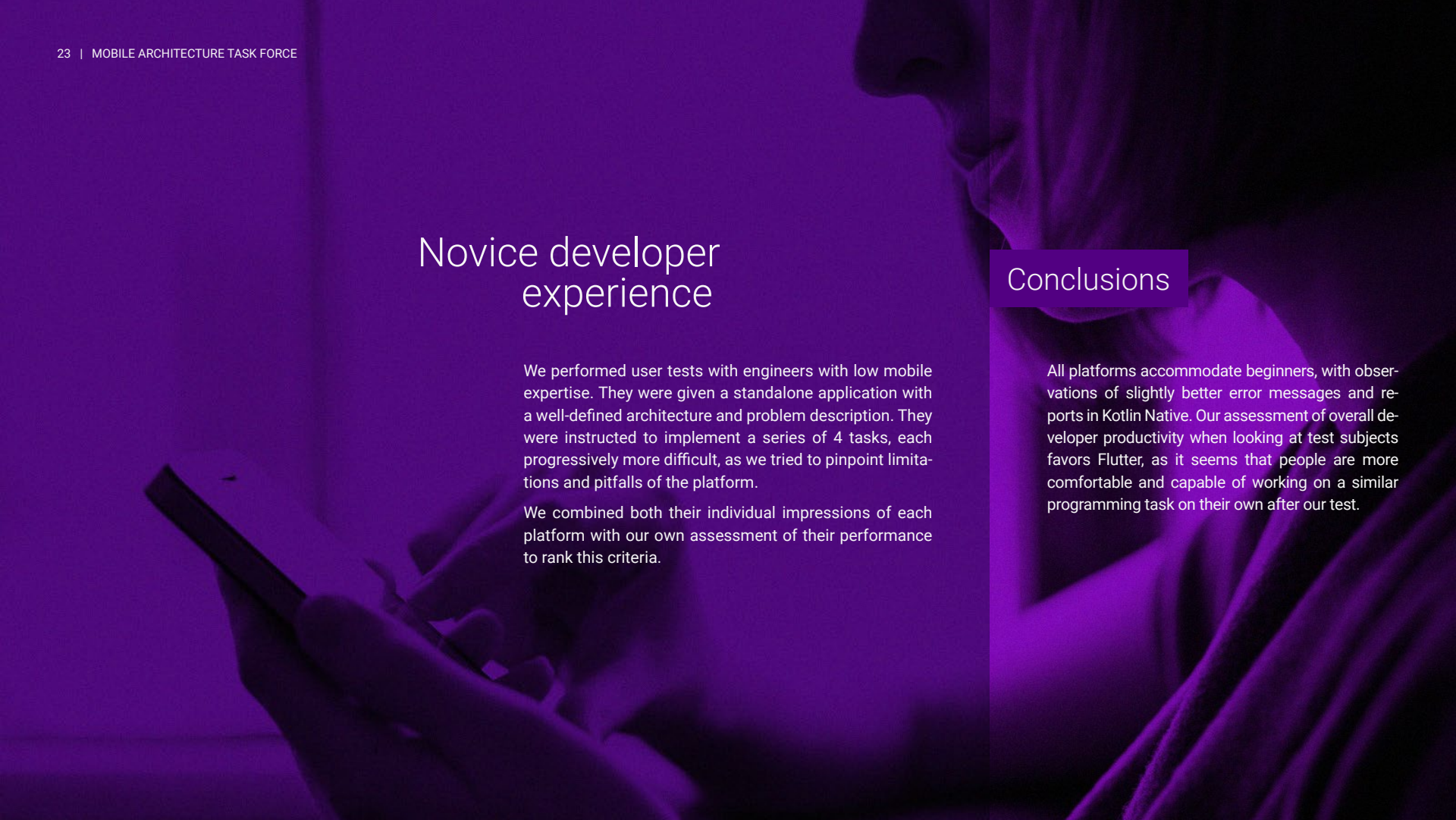
The Decision

To make the decision we realized we had too many criteria and decided to only consider the 7 most important ones.



Developer Experience





Novice developer experience

We performed user tests with engineers with low mobile expertise. They were given a standalone application with a well-defined architecture and problem description. They were instructed to implement a series of 4 tasks, each progressively more difficult, as we tried to pinpoint limitations and pitfalls of the platform.

We combined both their individual impressions of each platform with our own assessment of their performance to rank this criteria.

Conclusions

All platforms accommodate beginners, with observations of slightly better error messages and reports in Kotlin Native. Our assessment of overall developer productivity when looking at test subjects favors Flutter, as it seems that people are more comfortable and capable of working on a similar programming task on their own after our test.

Skilled developer experience

User tests alone are not enough to have a good measure of developer productivity as it only takes into account the very first contact with the platform. We surveyed a board of advisors to get a sense of what other tools and qualities in the platform will experienced engineers look for. We gathered data for developer tooling and Build/CI infrastructure.

Tooling

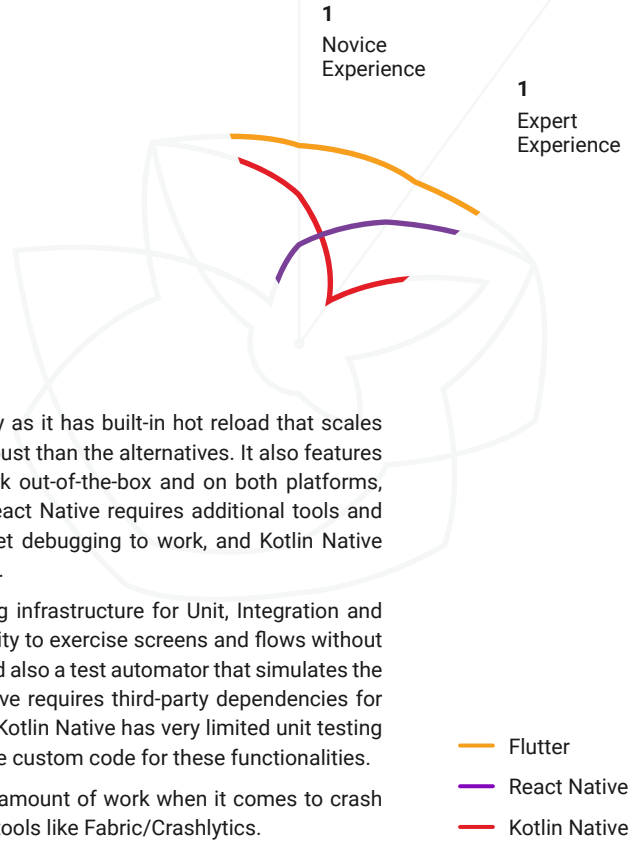
Platform	Reload at scale	Debugger quality	Dev env stability	Crash reporting	Score
Flutter	High	Logic - High UI - High	*High**	Works (native + Dart exceptions). Dart exceptions are non-fatal	Medium -high
React Native	Medium	Logic: Low UI: Medium	Medium	Works (native separate from JS exceptions)	Medium
Kotlin Native	Low	Logic Android: High (Native dev workflow) UI - Android: High (Native workflow) Logic iOS: Low UI - iOS: High (Native workflow)	*Medium**	Android: Identical iOS: not fully production ready, could be implemented by us, included in the roadmap	Low

*Gut feel Projection; No actual data was collected/available.

Build/CI

Platform	Reload at scale	Test stability	Testability	Score
Flutter	Medium	Medium-High	High	Medium-high
React Native	Low	Medium	High	Medium
Kotlin Native	Medium	Low	Low	Medium-low

Conclusions



Flutter seems to have better scalability as it has built-in hot reload that scales well with the codebase and is more robust than the alternatives. It also features code and UI debugging tools that work out-of-the-box and on both platforms, integrated with the IDE. In contrast, React Native requires additional tools and setup (such as using a browser) to get debugging to work, and Kotlin Native requires the underlying platform's tools.

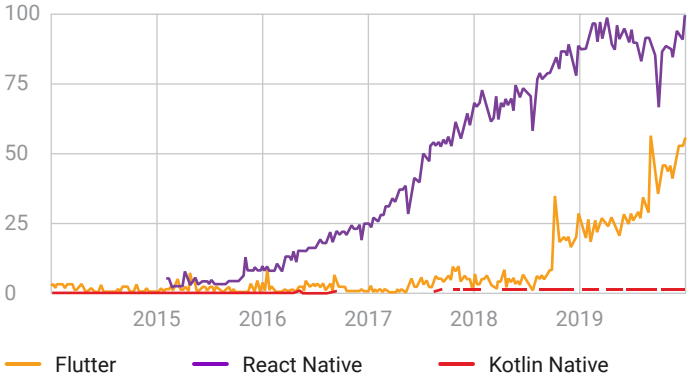
Furthermore, Flutter has built-in testing infrastructure for Unit, Integration and End-to-End tests. This includes the ability to exercise screens and flows without the need for rendering to the screen, and also a test automator that simulates the input of a user. In contrast, React Native requires third-party dependencies for integration and E2E tests to work, and Kotlin Native has very limited unit testing support, which would require us to write custom code for these functionalities.

Finally, all three platforms need some amount of work when it comes to crash reporting and integration with existing tools like Fabric/Crashlytics.

Long-term viability

The second most important criteria is the life expectancy of the framework, and whether it fits into our long-term vision. For our approach, we are considering 5 years as long-term, as this reflects the speed characteristics of our company and the how young mobile technologies are in general. We are also considering the likelihood of the community absorbing the maintenance costs in case the original maintainer drops support for it.

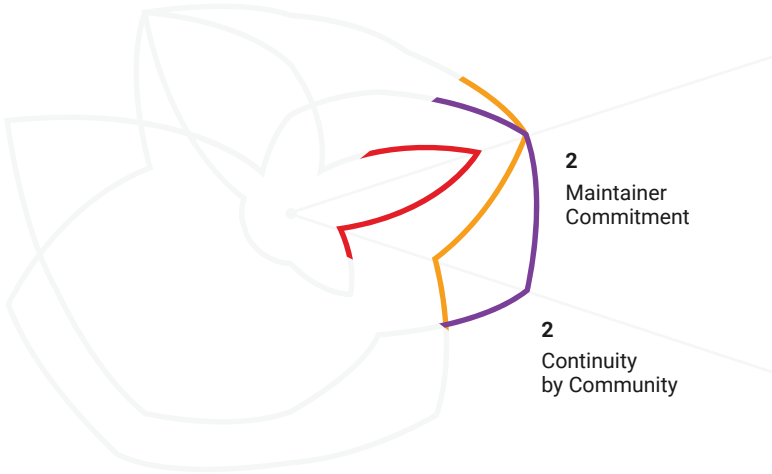
Platform search popularity



Platform	Community size		
	# Stars/Forks	Stackoverflow questions	Google trends
Flutter	56,960/6,167	12,630	Upwards trend
React Native	75,131/16,705	48,301	Stable since Jan 2017, still in 1st place
Kotlin Native	5,628/399	200	Undetectable on the Platform search popularity graph

Platform	Number of Contributors	Risk of being abandoned by maintainer	Continuity by community
Flutter	61% of last 1.000 PRs from Google [Excluding bots]	Medium-Low	Medium
React Native	9,5% of last 1.000 PRs from Facebook 9,4% of last 1.000 PRs from Core Contributors	Medium-Low	High
Kotlin Native	87% of last 1.000 PRs from JetBrains	Medium-High	Low

Conclusions



The community around React Native is larger, more mature and has been around for longer than all other alternatives. While Flutter is a more recent technology, it has better official documentation and support from the maintainer than React Native and is growing at an accelerated rate. Furthermore, the community in React Native is more engaged and more likely to continue development if Facebook decides to drop official support.

Regarding maintainer commitment, we see Flutter and React Native as being equally likely to have continued support. We've had direct contact with a Sr. Product Manager at the Flutter team who assured us that Flutter has been a multi-year project before initial public release, and that it is currently being used in more than 10 projects at Google, many of which will be public and span across millions to tens of millions of users.

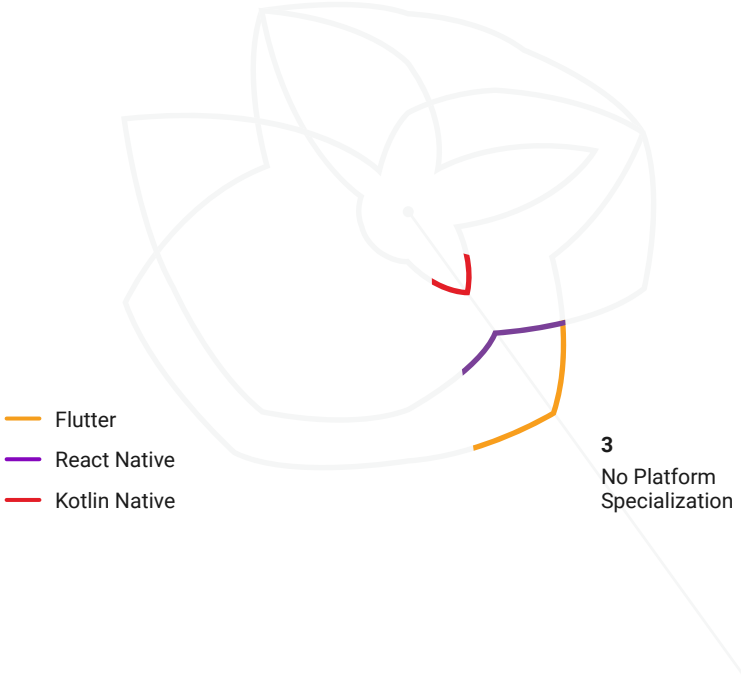
No platform specialization

The risk of requiring platform-specific code is proportional to how close the platform is to the underlying operating system. The further away, the bigger the abstraction, and the less likely it is for the cross-platform code to break in only one of the platforms.

Platform	Risk of requiring platform-specific code
Flutter	Low
React Native	Medium
Kotlin Native	High

Conclusions

Since Flutter renders directly to the screen, it is less likely than React Native to have OS-specific behavior in its UI elements and abstractions. Flutter also offers more built-in abstractions (such as navigation) which would otherwise require external dependencies to fulfill. In Kotlin Native, there is no provided UI abstraction and as such we would have to write our own. This increases the likelihood of leaking the internal workings of the native platforms.



Ease of incremental extension

The cost estimation is based on the effort required to add a brand new component to the Nubank Design System (i.e.: one that cannot be represented as a composition of existing components).

Incremental friction estimations are based on whether the same teams that are working on features are able to create new design system components, or whether they must necessarily rely on an external team of native specialists to develop them.

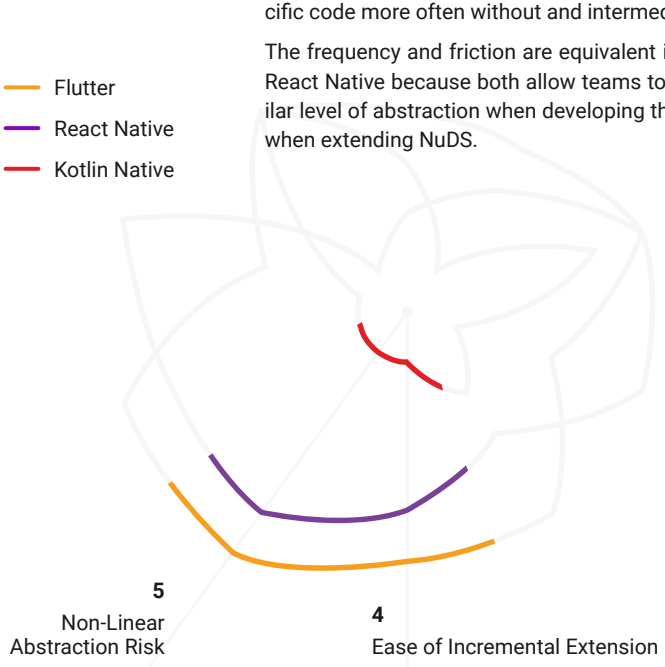
Note: The final score was translated to a scale where higher is better. For the intermediate columns, lower is better.

Platform	Cost estimation	Estimated Frequency	Incremental Friction estimation	Score
Flutter	1x	Low	Low	High
React Native	2x	Low	Low	Medium-High
Kotlin Native	10x	High	High	Low

Conclusions

The cost estimation is lower on Flutter because it absorbs more platform specificities than the others. React Native sits closer to Flutter because it does provide an intermediate layer where you can work on top of. Kotlin Native would be more costly as we would need to jump to platform specific code more often without and intermediate layer.

The frequency and friction are equivalent in Flutter and in React Native because both allow teams to work on a similar level of abstraction when developing their product and when extending NuDS.



Risk of non-linear cost

Note: The final score was translated to a scale where higher is better. For the intermediate columns, lower is better.

Platform	Risk of breaking UI abstractions	Score
Flutter	Very low	Very high
React Native	Low	High
Kotlin Native	High	Low

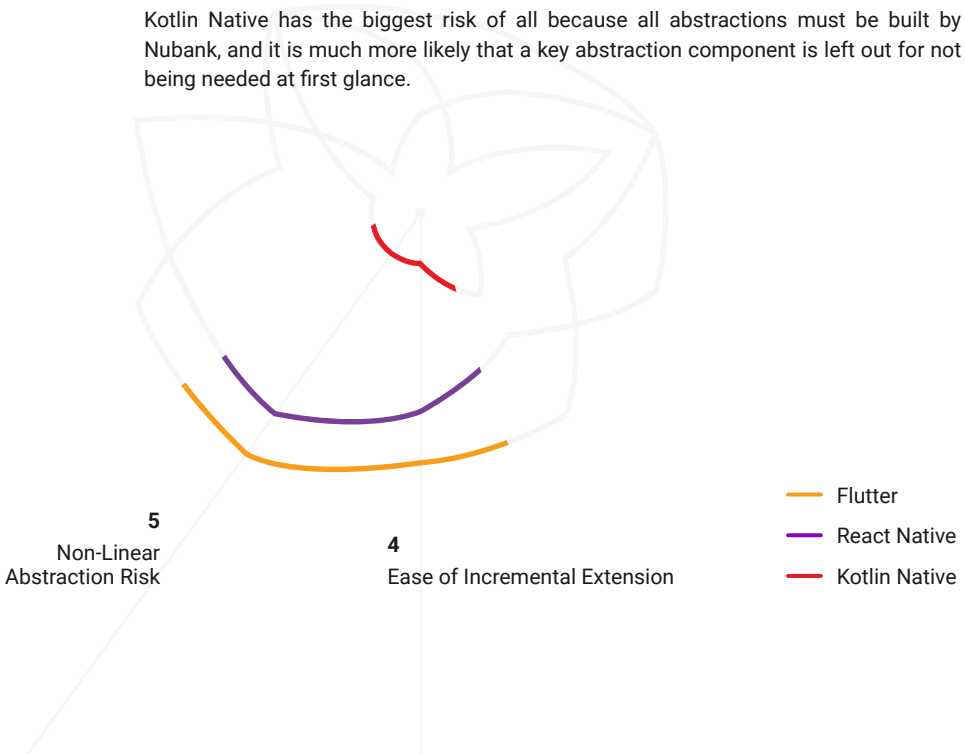
Flutter and React Native have lower risk because the whole generic UI abstraction is encompassed by the framework. React Native has to do extra work to make the abstraction fit both underlying OS, while Flutter dictates the abstraction.

Kotlin Native has a high risk because we would have to implement all the required code for feature teams to work on top of. Implementing a generic layout layer would entail a lot of work, but a small abstraction may fail to be easily extensible to new use cases.

Conclusions

Since Flutter has many built-in abstractions and components are composable by design, there is little risk that our NuDS abstractions won't fit into the provided model. React Native has a similar capability, but is somewhat lacking in built-in primitives, thus requiring either custom components or third-party dependencies. An example of this would be the collapsing header component, which is today a third-party dependency in React Native but is provided built-in in Flutter.

Kotlin Native has the biggest risk of all because all abstractions must be built by Nubank, and it is much more likely that a key abstraction component is left out for not being needed at first glance.



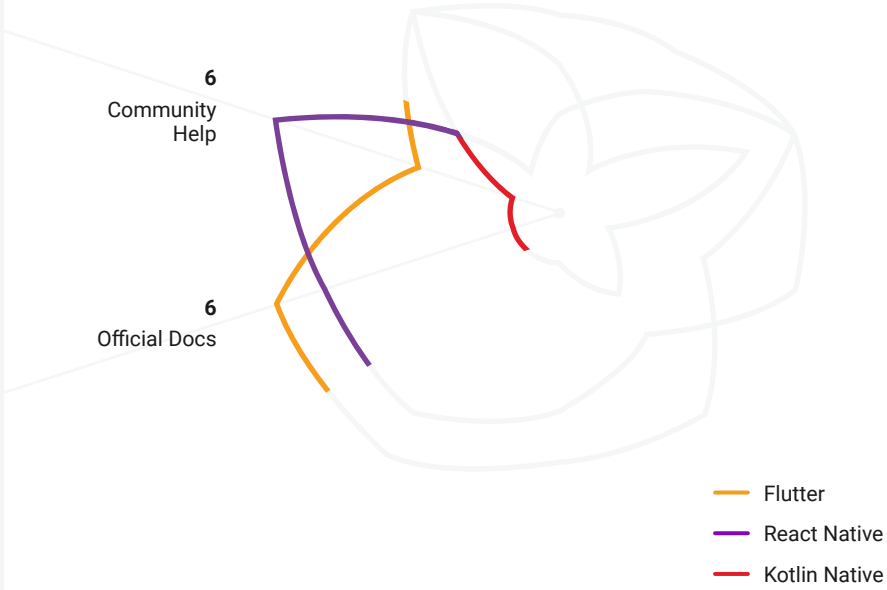
Available learning resources

Platform	Official Online Documentation	Examples and Answers on StackOverflow	Books, conferences and courses	Score
Flutter	Very Good	Medium	Low availability	Medium-High
React Native	Good	High	Medium availability	Medium-High
Kotlin Native	Poor	Very low	Very Low availability	Low

Conclusions

React Native's community is much larger and well established, and there are many more available blog posts, articles, courses, videos, etc. On the other hand, Flutter's official documentation is richer and broader, and there's also more text and video content produced by the Flutter team. We estimate these points to somewhat balance each other and thus consider both Flutter and React Native as having a similar score.

Kotlin Native's documentation is very limited, and most of the engineer's day to day resources would need to be created by Nubank. Further documentation would be required to represent the UI abstractions and available components.



Breaking changes

For this criterion we surveyed for the history of breaking changes in the platform, taking the API, available tools and development environment into account. We also considered the breaking changes in the main required dependencies.

Note: The final score was translated to a scale where higher is better.
For the intermediate columns, lower is better.

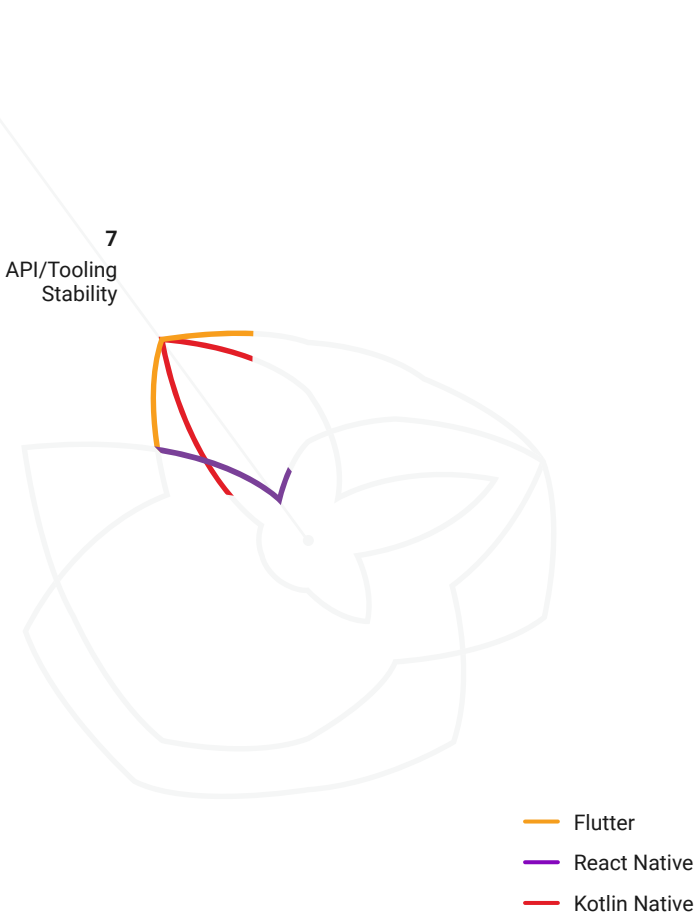
Platform	History of breaking changes on each platform	Breaking changes on dependencies	Score
Flutter	See changelog. No breaking changes since 1.0.0 (Currently 1.3.12, 26 releases since 1.0.0) See also Dart changelog for language breaking changes.	Very young, very few dependencies since it's more complete at its core	High
React Native	Pull requests are automatically marked with breaking change tag Looking at the changelog, we find ~2 breaking changes per monthly release.	Frequent, if we use the dependency it hurts	Low
Kotlin Native	Could not find API history See also Kotlin changelog for language breaking changes.	Very young, few dependencies	High

Dependency analysis

Platform	Production dependencies	Test dependencies
Flutter	3	2
React Native	32	22
Kotlin Native	7	5

Conclusions

React Native has an order of magnitude more dependencies than the other alternatives, and as such is much more vulnerable to breaking changes in those areas. Both Kotlin Native and Flutter have stable APIs, where Kotlin Native has a much smaller surface area



App stores restrictions

This criterion attempts to measure the stability of the platform regarding store distribution and potential restrictions that could apply to our app if a given platform is chosen. We considered the presence of Over the Air updates a potential risk, since Apple has restricted this behavior in the past. Another factor we took into account was the possibility of rejection due to the cross-platform UI framework not complying with the platform's built-in guidelines. This is more likely to happen in Flutter since it does not use the system's built-in components.

Note: The final score was translated to a scale where higher is better. For the intermediate columns, lower is better.

Conclusions

Platform	Technological Risk	Market Weight	Score
Flutter	Low (Apple might judge Flutter apps UX lacking)	Low (Google Ads, Alibaba)	Medium-High
React Native	Very Low (OTA possibility)	High (Facebook, Instagram, Uber Eats, Pinterest)	High
Kotlin Native	Minimum	Minimum	Very high

With respect to the possibility of being restricted by the store guidelines, Kotlin Native is the safest option since it does not have the built-in possibility to run code that wasn't already bundled with the application and it requires the creation of our own UI abstractions using system components.

React Native also has low risk, mainly because it provides infrastructure to receive Over the Air updates and that could be seen as a problem for the store maintainers.

Flutter is the only framework that does not rely on native components, and as such has new dimension of concerns associated with it, both regarding user experience and human interface guidelines compliance. Even though it is possible that App Store restrictions might come up that affect Flutter apps, we find this possibility is very unlikely today and assigning a Medium-High score to Flutter.

Capability limitations

Note: The final score was translated to a scale where higher is better.
For the intermediate columns, lower is better.

Platform	Look for customer-facing things that cannot be done with the cross-platform technology (only native)	Score
Flutter	State restoration on Android cannot in a synchronous way App extensions on iOS are not fully supported Note: Flutter is actively working on fidelity for iOS	Medium - Somewhat limited, need to wait for Flutter to implement new features; cannot be used in auxiliary processes
React Native	Can't build iOS native extensions (Today extension, Siri, ...) For the same reason as Flutter, RN's state restoration on Android should not work (unverified).	Medium-High - Able to use native components, but cannot be used in auxiliary processes
Kotlin Native	Does not apply since all UI is necessarily native	Very high - No limits

Considering capabilities beyond the mobile app, both Flutter and React Native go over the memory threshold established by the OS to create app extensions (Today extension, Siri extension, etc.). Kotlin Native has a much smaller memory footprint and thus can be used in these contexts.

Platform/OS parity

Platform	If OS changes at UI level, will the platform be behind?
Flutter	Yes. Addition of new UI components/functionality that cannot be accessed outside UIKit. (Autofill support) We/Google would either have to embed a PlatformView or replicate the behavior from the private API.
React Native	No. (Use the underlying system components)
Kotlin Native	No. (Use the underlying system components)

Both Kotlin Native and React Native do use the underlying OS provided components. Only Flutter has to either reimplement the same native behavior or embed a PlatformView.

Conclusions

Regarding platform limitations, Kotlin Native is the most flexible approach. It allows code sharing in all contexts and poses no limits on platform/device features.

Flutter and React Native have similar limitations mostly due to their memory footprint and not being usable outside the main application process. This limitation may change over time, but we consider that it is not a blocker for use in our app since we do not rely heavily on auxiliary process usage and it is still possible to write native code to fulfill these requirements.

Roadmap

In this criterion we attempted to investigate any potential problems for the future of each technology and whether it would diverge from our long-term interests. We found no contention points, and so the analysis was limited to the visibility we can have of the product's roadmap.

Platform	Roadmap information from each platform and verify with Nubank's expectations	Score
Flutter	Open roadmap with milestones and project tracking Adding to an existing application as a self-contained framework is a first-party flow is in the official roadmap (and almost done)	High
React Native	Broad roadmap goals which involve big refactors/ improvements and ownership transfers.	High
Kotlin Native	No public roadmap; Issue tracker with an internal tool	High

Conclusions

Although there are some small unresolved issues (and open PRs) with Flutter and React Native, those aren't blockers for us. Most have existing workarounds or would disappear when we adopt Flutter or React Native as the toplevel project.

Initial Abstraction cost

Note: The final score was translated to a scale where higher is better.
For the intermediate columns, lower is better.

Platform	# of Components to build	Effort to create CI pipeline	Effort to create test infrastructure	Score
Flutter	Thin wrappers only	Medium	Medium	Medium
React Native	Thin wrappers only	Low	Very low (already exists)	High
Kotlin Native	Thick wrappers (actual components) and layout abstractions	Medium	Very High	Very Low

React Native has the lowest initial cost because it is already integrated in the app. The initial cost would be only adding top level support to React Native in the app.

Flutter's proof of concept allowed us to estimate an initial cost as lower than React Native's original initial cost, but still harder to integrate than the existing React Native. The proof of concept was later used for User Testing and can be seen in Appendix A.

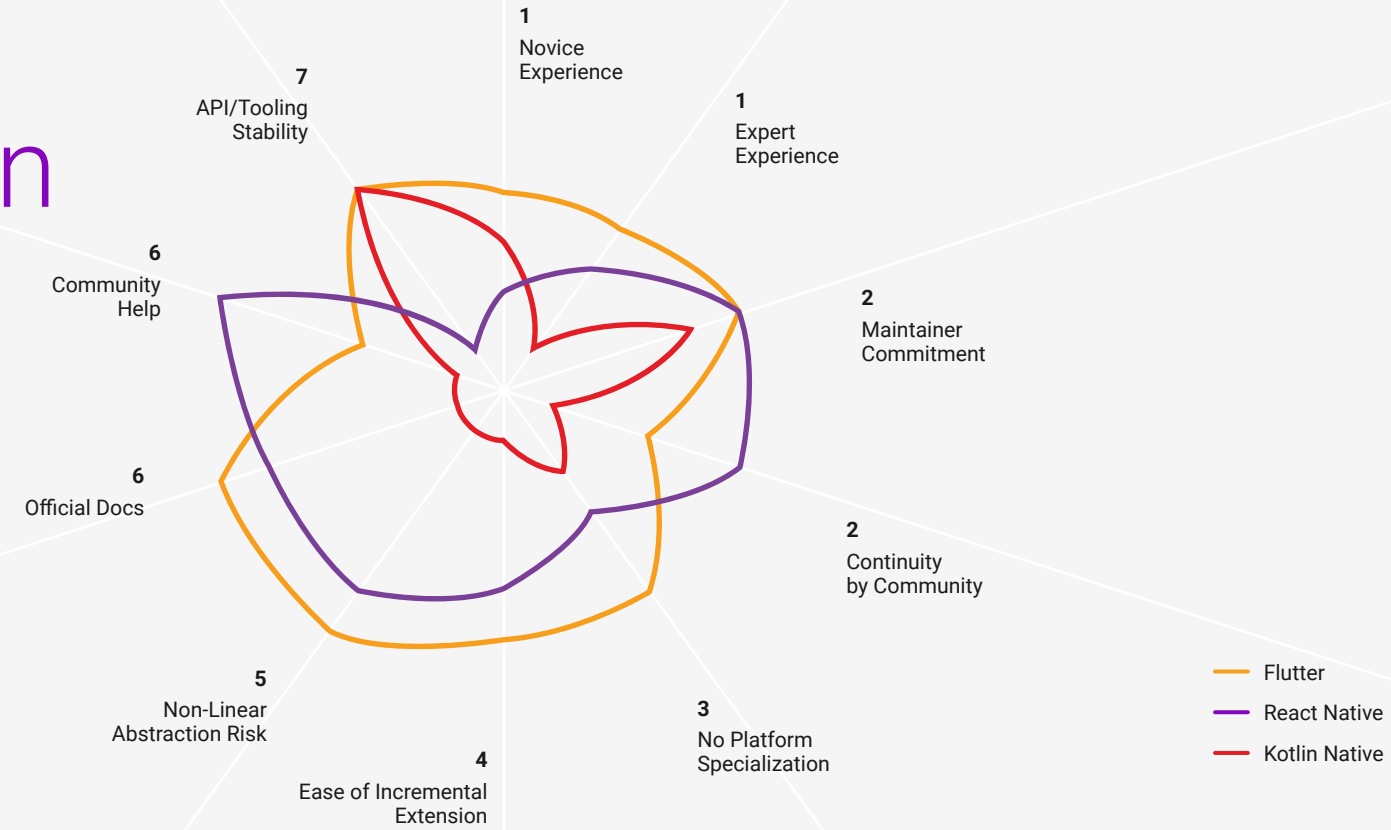
Kotlin Native has a high cost estimation because we would have to bootstrap the initial components on both platforms and also extend the toy test infrastructure created by us for the user tests.

Conclusions

The initial abstraction cost was intentionally deprioritized when compared to the other criteria in this document, mainly because it would introduce high biases to continue using existing technologies. Since these costs are present only once (and do not affect us on every feature or extension), we believe this is important to keep track of, but not a main driver for the decision.

The Decision

To make the decision we realized we had too many criteria and decided to only consider the 7 most important ones.



The Decision

We've decided to adopt Flutter as the future platform for mobile development at Nubank. We believe this to be aligned with the company's best interests and that it fits the development model and engineering culture our company follows.

Drawing from our own experiences (80% of our Android codebase is Kotlin, NuConta is developed in React Native) and evaluating our alternatives against Nubank priorities we feel like Kotlin is a great language to work with. But Kotlin Native is the only platform that doesn't provide a UI abstraction, making it dependent on native platform tooling for developing and testing. While it scored higher in our lowest priority criteria, not showing limitations of capabilities or risks for app store restrictions, we felt that especially when it came to testing support for expert engineers, Kotlin Native is not ready for us.

We feared a bias towards React Native, so we consciously lowered the priority of another criteria: the cost of building the initial abstraction on the platform, where React Native was a clear winner.

When looking at more important criteria, React Native also wins in community support. We felt no fear of the continuity and evolution of the project and were really happy with the amount of documentation and learning resources available. When it came to breaking changes however, we found that React Native has an order of magnitude more dependencies than the other alternatives, and as such is much more vulnerable to maintenance and upgrading pains.

Our engineering culture strongly encourages test automation, so Flutter shined with its great testing capabilities, that fit nicely with our mindset (built-in testing infrastructure for Unit, Integration and End-to-End tests without the need for rendering to the screen). Whereas React Native requires third-party dependencies, which makes it more prone to breaking changes. We found the Flutter development experience to be superior, with better hot reload capabilities, very strong official documentation, and a more stable API.

After a lot of discussion and contention up to the last minute we decided to use Flutter as Nubank's main technology for mobile development. This means the new features will be written in Flutter and as the product evolves we expect it to become the greater percentage of our codebase.

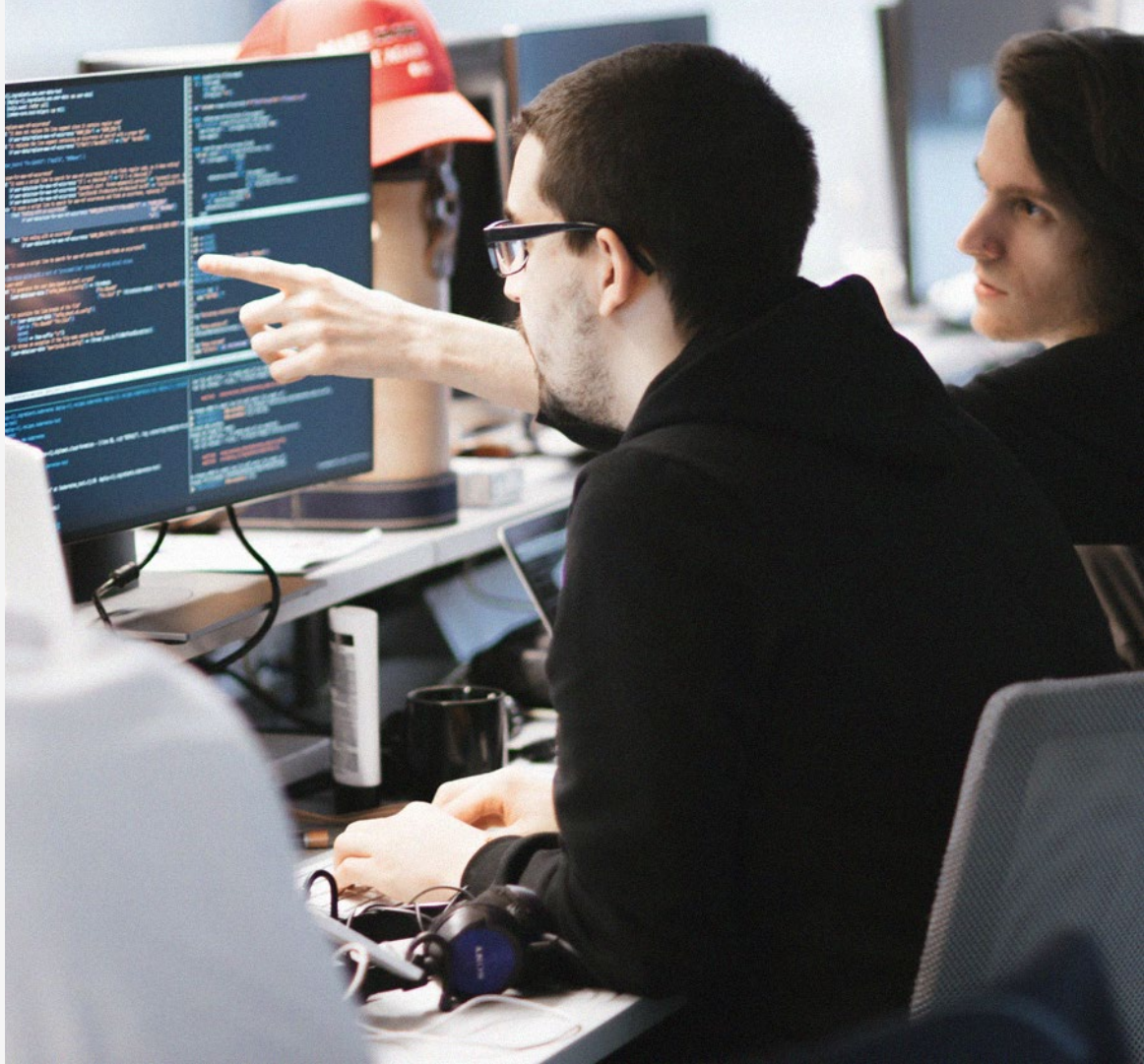
Conclusions

So far it's been great to use Flutter, we expect to have more features built or migrated to Flutter out to our users very soon.

Having to include Flutter in a running app with millions of clients comes with its own set of challenges that we're gradually overcoming, the first of them being:

- changes in build pipelines,
- creating the main platform channels,
- integrating routing amongst React Native, Flutter, Kotlin and Swift so we can maintain interoperability.

While Flutter is going to be our main technology, native developers are still needed and valued as each platform has its own set of features that require native code (e.g.: native plugins like GPS and camera, Apple Watch, Android minimized apps, etc...) and as the software engineering team at Nubank grows individual specialization is welcomed.



APPENDIX A

Mobile Architecture User Testing

Installation instructions

Follow the installation guides from the platform you want to test:

Flutter

React Native

> CLI Quickstart

Kotlin Native

Sync the Gradle project after opening the Kotlin Native sample app

The sample apps are hosted in the mobile-architecture-taskforce Github project. (Link removed. This is a private repository, with the code needed to perform the tasks)

Instructions for running the applications are on the repository readme:

Flutter

$ cd Flutter/flutter_mob_arch_tf/
Run F5 in VSCode and pick the iPhone or Android emulator.

React Native

$ cd React-Native/RNMobArchTF/
$ react-native run-ios

Or

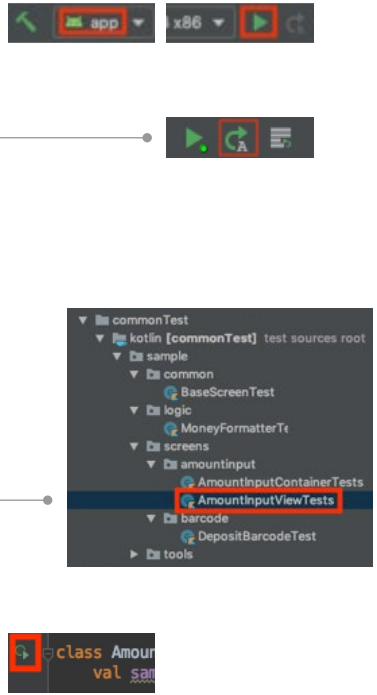
$ react-native run-android
Then open VSCode for editing the TypeScript code of the app.

Kotlin Native

Run the app module on Android Studio 3.5 Preview

Environment

	Flutter	React Native	Kotlin Native
IDE	Visual Studio Code/IntelliJ	Visual Studio Code/IntelliJ	Android Studio Preview
Command line executable	Flutter	react-native	./gradlew
Running the project	Visual Studio Code: Debug > Start Debugging (F5) IntelliJ: Run > Run 'main.dart'	react-native run-android	Android Studio: Select the app module and press run You may experiment with hot reload by using the Apply Changes button
Running the tests	Visual Studio Code: flutter test [filename] IntelliJ: Play button on the side of the test name	npm test [filename] [--watch]	Select a test class Run a test



Sample App

The sample application will implement the flow to deposit money into their NuConta using “boleto”, with the following screens:

Screen 1 (User Input)

Amount Input

Screen 2 (Result Screen)

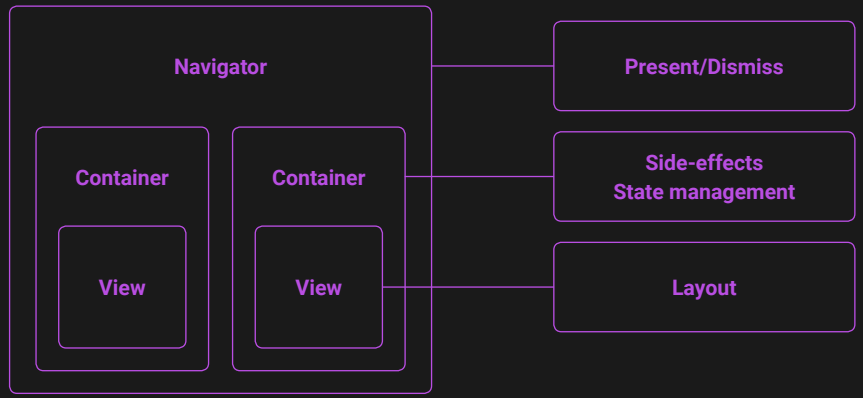
Barcode

Architecture

Responsibilities are divided among several classes to ensure each class is specialized and does only one job.

You will encounter the following classes for handling UI:

– **Navigator** – **Container** – **View**



Navigators are responsible for presenting/dismissing containers:
- It knows the order between containers
- It issues commands to navigate imperatively
- It adapts/transfers data between containers

Containers represent screens. They wrap layout-only views:
- Manage the view state
- Executes side–effects based on interactions
  - Perform requests
  - Copy to clipboard
- Exposes output from screens (i.e.: data to be used elsewhere in the flow)

View is pure layout and relaying of interactions.
- It positions components on the screen
- Exposes interactions using callbacks

Example

Presents containers
Pass data from containers

Navigator

Enabled state
Validation logic
Request to generate barcode

Container

Color/font component positions
Button press detection
Enabled

View

Tasks

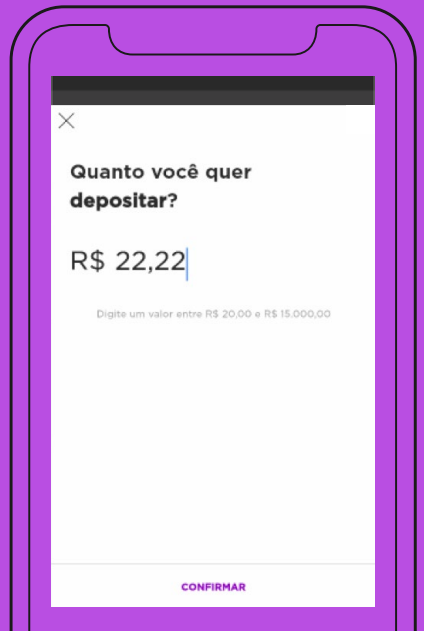
1. **LAYOUT:**
   UI tweaks
   a. Move the “VENCIMENTO ...” text in the Barcode screen from below the amount to above it.
   b. Change the text to be “DATA DE VENCIMENTO ...”

Expected result:

2. **NAVIGATION:**

   a. Navigate back to the Amount Input screen when the amount is tapped in the Barcode screen.
   b. Create a test that verifies this behavior

Expected result:



Tasks

3. **FUNCTIONALITY:**
   Add shortcut buttons with common values on Amount Input (i.e.: R$20, 50, 100)

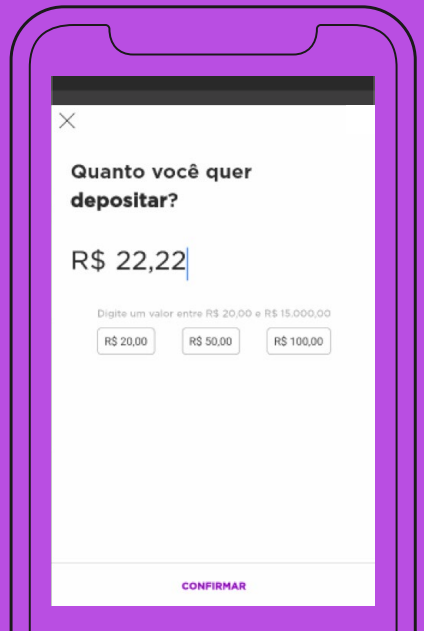
   a. Add 3 instances of ShortcutButton (this class is provided) to the Amount Input screen one beside the other

   b. Make each button lead to the next screen with the selected amount

   c. Add tests to assert that each button navigates to the following screen with the proper boleto amount

Note: *Don't worry too much about spacing and positioning since they may vary depending on the platform. The important aspects are the horizontal arrangement of the buttons, the consistent spacing between them and being centered horizontally.*

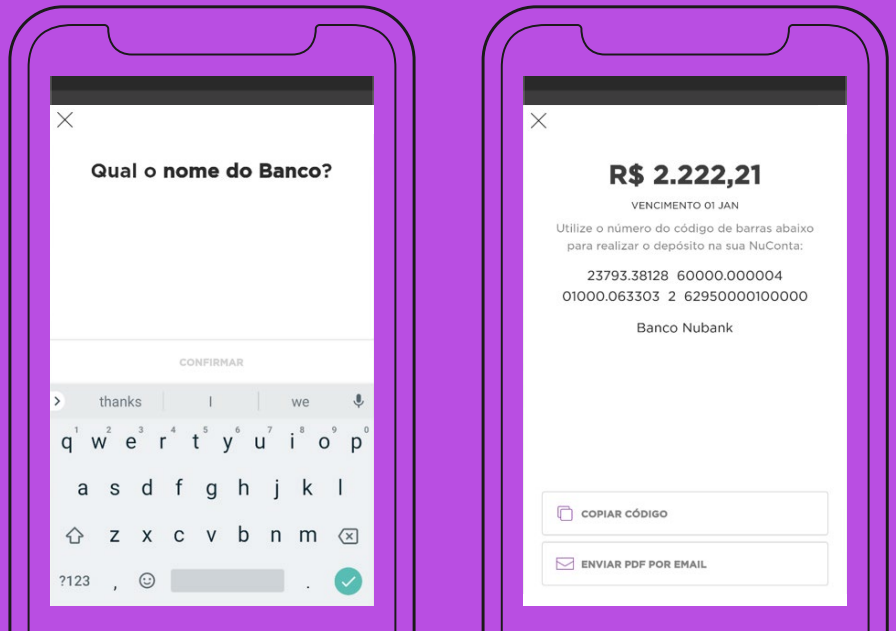
Expected result:



4. **NAVIGATION:**
   Add a new screen in between the existing two that asks for the bank you are depositing from using a text input field.
   a. Create the new screen with a text input
   b. Navigate to the new screen after Amount Input
   c. Get the value from the text input and display it on the last screen, below the readable barcode.

Note: *Don't worry about the bold text parts; NuML would be the preferred way to do this but it is not implemented in all platforms.*

Expected result:



APPENDIX B - SURVEY

Mobile Architecture Taskforce Survey

Mobile Architecture Taskforce Survey

This survey is to be filled after completing the Mobile Architecture platform evaluation test

What's your name?

Comfort

How was your experience during the test?

What platform were you using?

- ○ Flutter
- ○ React-Native
- ○ Kotlin-Native

How comfortable did you feel using this platform?*

	1	2	3	4	5	
Couldn't figure out how to do anything	○	○	○	○	○	Instantly knew how to do everything

How comfortable were you with your IDE?

	1	2	3	4	5	
Didn't know how to do anything	○	○	○	○	○	Very, knew all the shortcuts and tools

How readable were the error messages you encountered?*

	1	2	3	4	5	
Didn't know how to do anything	○	○	○	○	○	Very, knew all the shortcuts and tools

Productivity

Subjectively, how productive did you feel throughout the exercise?

	1	2	3	4	5	
I was constantly stuck and didn't know what to do	○	○	○	○	○	It was a lot easier than I expected to complete the tasks

How easy was it to understand the architecture/organization of the codebase

	1	2	3	4	5	
Couldn't find anything	○	○	○	○	○	Found everything with ease

Learnability

Did you have any difficulty reading the language?

Where reading means understanding the details of the language, what are loops, variables and the logical structure.

- Yes
- Somewhat, but I managed to understand after some time
- No

Did you face any problem you couldn't solve even after searching online?

- Yes
- No

If so, what was the problem?
[Leave blank if none]

What did you think about the quality of the answers available online?

	1	2	3	4	5	
Worthless	○	○	○	○	○	Solved all my problems

How would you qualify the learning curve of this platform?

	1	2	3	4	5	
Very steep, hard to get started	○	○	○	○	○	Very incremental, easy to learn more each step

How prepared do you feel to perform a similar task?

	1	2	3	4	5	
As if starting from scratch	○	○	○	○	○	Very comfortable

October 2019