SAVERY: A Framework for the Assessment and Comparison of Mobile Development Tools

José Donato, Naghmeh Ivaki, and Nuno Antunes
CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
jose.donato@openbb.finance, naghmeh@dei.uc.pt, nmsa@dei.uc.pt

Abstract—Different types of mobile development tools and frameworks to build web applications are constantly and rapidly emerging, making it difficult for developers to choose an appropriate one for their needs. In this work, we propose a novel framework to assess and benchmark development tools according to performance (and potentially dependability and security). The framework defines the components and procedures required to define and execute the benchmarks. To demonstrate the applicability of the proposed framework, we implemented it for a concrete benchmark focused on assessing the performance of a representative application developed by different development tools. For this, we defined a representative set of features for the application and a set of relevant metrics to characterize their performance.The benchmarking campaign is executed on each application in an automated manner. The metrics are collected during the execution of the campaign and the results are then analyzed to compare the applications, their features, and the development tools that produced them. The results show that the same applications implemented in different frameworks are different in terms of software metrics and build time and show different performance when executing different functions (features) in terms of computation time, memory, and CPU usage. This reveals the effectiveness of the approach and the framework proposed to assess and compare the mobile development tools.

Keywords–Mobile applications; Native applications; Web applications; Benchmarking; Performance; Android

1. INTRODUCTION

During the last few decades, smartphones have improved tremendously, and their usage has come along (i.e., about 6.9 billion mobile phone in 2023). This massive adaptation led to continuous improvement of mobile devices (e.g., regarding their functionalities and computation power) and, consequently, increased mobile application requirements. As the requirements for this type of application have been expanded to cover more functionalities and achieve better quality, an increased number of **development tools** were created and are available to support developers in creating such applications.
In general, three main types of mobile applications can be developed: **Native applications**, which run natively after being installed. They are typically the primary choice for developers because they are fast and work offline. However, when the ability to support multiple platforms is vital, development and maintenance become time-consuming as they are tied to the target platform [28]. **Web applications**, which run directly on a browser, and thus, they can run on all devices using the same codebase. However, sometimes they are not able to achieve the same performance as native applications or provide offline functionalities, although these limitations are being reduced by Progressive Web Applications [5]. **Hybrid applications**, which run natively, but they usually are built using web technologies and packed into native containers. They enable the development of native applications for multiple platforms, sometimes at the cost of performance [18].
When developing an application for mobile devices, developers need to choose an appropriate approach and an adequate development tool (or framework) to implement it. The main associated challenge is that there are countless options with different characteristics (e.g., React Native or Kotlin for native, React.js or Svelte for web). In the JavaScript ecosystem, this number is constantly increasing, causing "JavaScript Fatigue" [17]. Thus, there is a clear need for techniques that allow assessing and comparing different properties of mobile development tools, both the ones currently available and the ones that are yet to appear, to help developers decide which are the best solutions based on their requirements.
This paper proposes a new framework for assessing different mobile application development tools considering several properties, such as performance (and potentially dependability and security). To implement and evaluate the proposed framework, we designed and implemented a concrete benchmark to evaluate and compare multiple popular development tools according to performance. As a direct comparison of the development tools (or frameworks) is nontrivial and does not precisely reflect the performance of the applications implemented by these tools, we propose **implementing an application, with a representative set of features, by using different development tools**. We defined a set of **8 principal features (reference app specification)** that users expect to find in a mobile application, such as access to the camera and rendering user's geolocation (see Section 4-A1). Then, we selected a set of 7 recent and widely used development tools including **React.js**, **Svelte**, **Next.js**, **Gatsby**, **Preact**, **React Native**, and **Ionic** (see Section 4-A2). Each tool was used to develop one application with the reference set of features, resulting in a set of similar applications in terms of functionality. We then selected several key **performance metrics** (e.g., RAM consumption, response times, CPU usage) to be measured during the tests.
All the applications should pass a set of automated func-

tionality tests specific to the implemented features (see Section 4-B2). Then, all applications are submitted to an automated assessment procedure during which metrics are collected and used for the (indirect) comparison of the development frameworks. Although all the applications were developed in-house, the benchmark is designed to welcome and incorporate future improvements. This is an open-source project, which is called **SAVERY** and is inspired by Techempower Benchmarks [24]. The project is open to the contribution of the community.

The main contributions of this study are as follows:

- Proposing a new and extensible framework, namely SAVERY, for assessing and benchmarking mobile development tools;
- Implementing the proposed framework to assess and compare 7 mobile development tools according to performance;
- Making this implementation available online for the community to use (and improve);

The results show that tools such as Preact or Next.js, which are built with performance as the primary requirement, show more promising results than other development tools (e.g., the ones based on React). It was also possible to observe that different tools performed differently in executing different features (or functionalities), highlighting the importance of helping developers choose an appropriate development tool in a clear and systematic way. We also observed that some tools performed better in terms of execution time but consumed much more memory and CPU. This is very relevant in an environment where resource and energy consumption are relevant. In general, this experiment revealed our benchmarking framework's effectiveness in assessing and comparing different development tools.

2. Background and Related Work

This section reviews the relevant concepts and related work in the context of this study.

2.1. Benchmarking

Benchmark can be defined as "standard tools that allow evaluating and comparing different systems or components according to specific properties (e.g., performance, dependability, security)" [29]. This allows customers and vendors to assess and compare different products to ease the process of selecting a product or to help to improve the products. In order to make a fair and meaningful comparison, a benchmark must be highly representative: i) the conditions and setup in which the benchmark is performed must be representative of realistic scenarios, ii) the properties (e.g., performance, security) of the product that are the target of the benchmark must be representative of the main functional and non-functional requirements of the product, and finally iii) the metrics chosen must be informative of those properties [12].

Four key components of a benchmark are: i) **System Under Benchmark (or SUB)**, which refers to the product that will be assessed in the benchmark; ii) **Rules/procedure**, which specify what needs to be followed during the benchmark campaign. The procedure must be easy to follow because the benchmark needs to be easily re-executed; iii) **Workload**, which refers to input values determining the type of operations that should be executed during the benchmark; and iv) **Measures/Metrics)**, which refer to the metrics or measurement tools to be used for assessment.

2.2. Evaluation and Benchmark of Mobile Development Tools

Evaluating web applications is not a new topic [11]. In 2008, a benchmark for Web 2.0 websites was published [22]. In this benchmark, a set of automation tools to generate workload and measure the websites' performance are presented. They developed two similar applications providing the same functionalities in PHP and Ruby on Rails. Similar to the previous work, in 2011, a study provided an architecture for benchmarking frameworks to develop Web 2.0 applications (e.g. Ruby on Rails or PHP) [2]. Another study compares a PWA with a traditional Web Application [23]. They used the same template for both applications and implemented the PWA features in one of them. This resulted in two identical applications: one that had PWA capabilities (e.g. work offline) and another one without it. In 2017, the energy efficiency in PWA was studied [19].

TechEmpower benchmark [24] measures the performance of multiple operations (e.g., JSON serialization, database queries, etc.) in traditional web applications. Although being considered a standard for benchmarking web applications, it only focuses on traditional web applications. The same applies to Web frameworks benchmark [25]. js-framework-benchmark [15] is another web application benchmark that consists of multiple operations around an HTML table developed using different SPA frameworks. However, the features in the application tested are not representative of what a SPA is capable of (i.e., CRUD operations around a table may be one use case of a SPA, but it is not the only one).

A study focused on Flutter created a framework to develop multiple applications from the same codebase, and compares them to Apache Cordova (now called Ionic) and Native [9]. A work performing performance analysis of a fully functional mobile application implemented in several **cross platform tools** and native for Android, iOS, and Windows Phone operating systems are presented in [28]. A comparison between a Native Android Application and a PWA was published in 2017 [6]. They developed the same app in both native Android with Java and PWA with React.js. There are more similar studies in the literature [1], [4], [8], [7].

In general, all studies provide theoretical explanations of the frameworks under study but have limitations when it comes to benchmark campaigns. Some benchmarks are out of date [20], [28] and some are limited to a small and unrepresentative number of features [6], [1], [25], [15].

3. Framework to Assess and Compare Mobile Development Tools

Comparing development tools is a challenging proposition, as it is not feasible or relevant to compare them in direct terms,

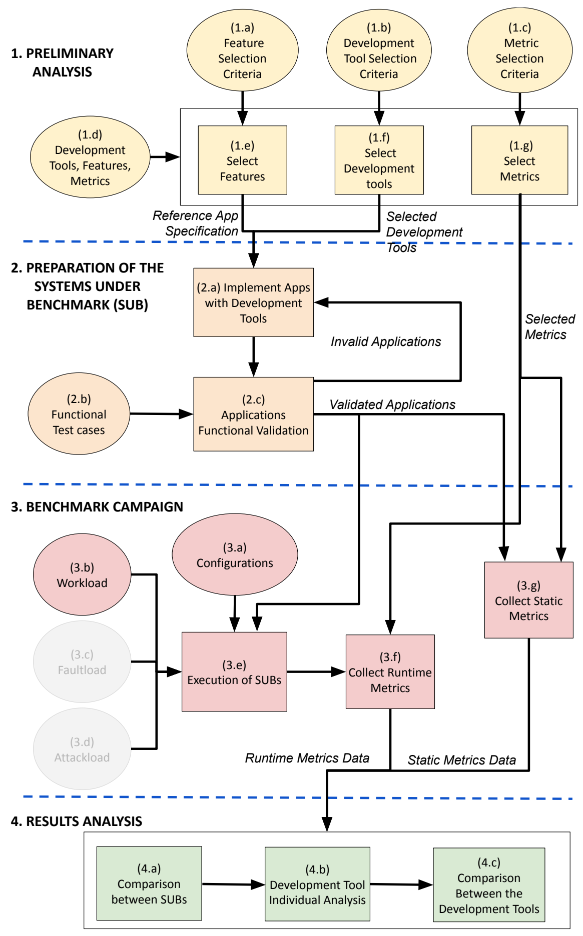
1. PRELIMINARY ANALYSIS
(1.a) Feature Selection Criteria
(1.b) Development Tool Selection Criteria
(1.c) Metric Selection Criteria
(1.d) Development Tools, Features, Metrics
(1.e) Select Features
(1.f) Select Development tools
(1.g) Select Metrics
Reference App Specification
Selected Development Tools
2. PREPARATION OF THE SYSTEMS UNDER BENCHMARK (SUB)
(2.a) Implement Apps with Development Tools
Invalid Applications
Selected Metrics
(2.b) Functional Test cases
(2.c) Applications Functional Validation
Validated Applications
3. BENCHMARK CAMPAIGN
(3.a) Configurations
(3.b) Workload
(3.c) Faultload
(3.d) Attackload
(3.e) Execution of SUBs
(3.f) Collect Runtime Metrics
(3.g) Collect Static Metrics
Runtime Metrics Data
Static Metrics Data
4. RESULTS ANALYSIS
(4.a) Comparison between SUBs
(4.b) Development Tool Individual Analysis
(4.c) Comparison Between the Development Tools

Figure 1. Overview of the proposed framework.

but instead, the usefulness of comparing them is to learn, which helps to create better applications or helps to create applications with less effort or knowledge from the developers. We argue that it is possible to make this comparison through the output of these tools, i.e., the developed applications. With this approach, one can evaluate and compare several kinds of properties such as performance (i.e., response time, CPU, and memory usage), dependability, security, and ease of development of the resulting application.

However, this analysis cannot be performed in an ad-hoc or naive manner. Thus, we propose a framework that carefully lays out the relevant components and procedures to develop benchmarks that can support this type of evaluation in a fair, useful, and representative way.

Figure. 1 depicts an overview of the key components of the framework. It details steps to define benchmarks that have specific targets in terms of applications and quality attributes. This framework is designed to be extensible and open to the community. There are four key phases, which are discussed in the ensuing paragraphs.

3.1. Preliminary analysis

This phase defines the benchmark's target, types of development tools to be adopted, and quality attributes of interest. To do so, we need to specify the **feature selection criteria**, the **development tool selection criteria**, and the **metric selection criteria**.

The main feature selection criterion is regarding the representativeness of the features. Based on this criterion, we analyze and select a representative list of features of mobile applications, which represents the set of features that the applications developed by the SUBs should support and should correspond to classes of applications that might be of interest to groups of developers. The result is a **reference app specification**, which is a specification that details which functionalities should be implemented in each application and the respective interactions, and a set of functional tests to those functionalities.

Next, we need to analyze the existing development tools for mobile applications to select a representative list of them (i.e., commonly used and recommended by the community). There are only two requirements/criteria that restrict the available development tools: i) need to produce applications for mobile devices, and they can be either native, web, or hybrid; ii) need to provide the ability to identify each element of the application with a unique id. This is crucial because, in tests, we treat the applications as a black box system and the only way to perform operations around the elements is if they have unique identifiers.

Finally, we need to select adequate metrics to collect. The metrics should help to understand the quality attributes of interest and to provide insightful comparisons. Two main types can be used:

- Static metrics: collected before running the campaign tests. Includes, for example, application size, line of codes, dependencies number (and dependencies number only for development), and build times.
- Runtime metrics: collected during the execution of SUBs (i.e., during the execution of benchmark campaign test cases) and includes, for example, response times, CPU usage, and RAM consumption.

The output of this phase (i.e., selected tools, selected features, and selected metrics) is used in the following phases.

3.2. Preparation of SUBs

In this second phase, we prepare the systems under assessment for the campaign that will be executed in the next phase. This phase is responsible for developing the applications following the reference app specification defined in the previous phase. After implementing the applications, they pass through a functional test (by submitting a predefined workload, i.e., input values that emulate and automate the applications' end-users' actions). The functional tests aim at validating the correct functioning of the applications.

Ideally, applications should be developed by developers that are fluent in each specific development tool to make sure that the implemented application adopts the corresponding best practices. When this is not possible, bias problems should be mitigated with the possibility of reviewing and improving the implementations or having multiple versions for the same application.

The tests to be developed should be as **non-intrusive** as possible. Therefore applications should be treated as black boxes during the execution of tests. As opposed to other studies in the area, there is no custom code in each application to collect response times during the execution of the tests, for example. Also, the benchmark needs to be easily reproduced, i.e., easy to repeat. We decided to create a set of automated tests (described as workload and more detailed below) that would always behave in the same manner without the need for human interaction. Hence, there is a need for a tool capable of producing automated tests to produce and run the workload.

We suggest including at least three elements in the experimental environment (i.e., the environment required to run the tests):

- Testing and Measurement Tool: responsible for a running tool that will send the automation tests to the target device and for gathering the previously defined metrics;
- Target device: where the applications under test will run. This device must be connected to the Testing and Measurement Tool;
- Remote server: serves the applications and the backend that the applications may require. This server should be deployed on a machine with a decent amount of RAM and high-speed network transfers to prevent it from becoming a bottleneck.

A benchmark **workload** refers to the applications' input values, which determine the type of operations that should be executed during the benchmark. The main associated challenge is workload representativeness. Our workload for both the validation and campaign phases is the same. It has the goal of emulating what an end-user of our application would do and comprises a set of automated tests that cover all the application functionalities in a certain sequence. The benchmark workload should follow the reference app specification in the first phase, i.e., it should go through the application components and test all of its functionalities.

A **Testing and Measurement Tool** should be implemented to create and submit the workload to the applications that will be running in the target device. Although in the validation phase, the metrics will not be collected, this tool also implements this process. Before proceeding to the next phase, all the applications must be validated by running the functional tests. Once all the applications are validated, they can proceed to the following phase.

3.3. Benchmark Campaign

This phase is responsible for executing the benchmark and collecting the metrics. As inputs, it receives the validated applications and the selected metrics. For certain configurations, the validated applications characterized now as System Under Benchmark (SUB) will be tested again several times while collecting the metrics with the help of the Testing and Measurement Tool (TMT).

3.3..1. Configurations: The campaign must comply with a certain set of configurations before the execution, such as:

- If the applications under test are web-based, they must be served with HTTPS.

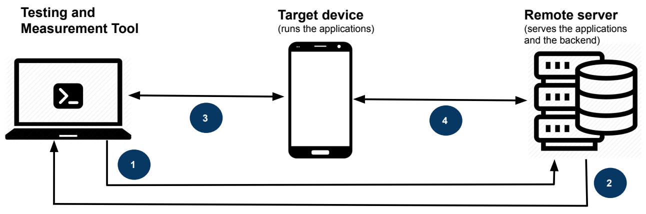
Testing and Measurement Tool
Target device
(runs the applications)
Remote server
(serves the applications and the backend)
1
2
3
4

Figure 2. Proposed Benchmark Campaign flow.

- Applications must be isolated from each other in the remote server to have fair comparisons.
- Permissions are not granted to any application. This is done to emulate what happens in real-world scenarios.
- The target device that is running the application cannot have background processes running.

3.3..2. Execution of subs: A campaign flow proposed to execute the SUBs is described in Figure. 2 and enumerated below:

1) TMT initiates the execution by requesting the application (SUB) to the remote server.
2) TMT receives and starts the connection to the target device that will run the SUB requested.
3) TMT connects to the target device and sends the SUB and the workload to the target device while periodically querying the previously defined metrics. TMT saves the results as it receives these values.
4) Target device may communicate with a backend in the remote server to perform some kind of operation required by the reference app (e.g., authenticate, fetch content from the database).
5) TMT finishes injecting the workload saves the results, closes the connection to the device, and cleans its usage.

The benchmark campaign should run several times for each application to reduce/eliminate the effect of random errors.

3.3..3. Collecting metrics: The Testing and Measurement Tool should also be implemented in the process of collecting the carefully chosen metrics. This process must: i) be reproducible, i.e., easy to execute since the campaign must be executed several times; ii) be non-intrusive: the goal is to minimize the impact of this process on the applications to have transparent results; iii) be fair to all types of applications under test (e.g., should not be more expensive to collect metrics in Web applications than in Native ones). When in need of an external library to count, for example, the lines of code of the applications, one should seek, when possible, libraries that support all the applications under test, i.e., avoid using different libraries to collect the same metric. Also, collecting metrics should as simple and naive as possible keep the campaign complexity low. For example, if all the applications are JavaScript-based, to collect the dependencies number, this process can be as simple as analyzing the `package.json` file. When possible, tools that are already available should be used to the detriment of adding new libraries (thus, increasing the campaign complexity). For example, if the target is an Android device, Android Debug Bridge (ADB) commands can be used to query important metrics such as CPU usage, RAM consumption, and battery levels, among others.

3.4. Result Analysis

Finally, the fourth phase, **Result analysis** concludes our assessment process. In this last phase, all the metrics are received from the previous phase. The metrics allow evaluation of each application and its development tool, and consequently, comparison between them. The Testing and Measurement Tool must save the results in a well-organized manner. Each campaign execution should output the following data: i) config: configuration used for the current campaign; ii) measures: list of measures gathered during the workload injection (e.g., CPU usages); iii) results: additional information about the campaign (e.g., start-up duration); iv) tests information: e.g., outcome, duration, timestamps.

4. A Performance Benchmark for Mobile Development Tools

To demonstrate the applicability of the proposed framework, we instantiated it in a concrete benchmark that targets entertainment and utility applications, development tools that are popular in the JavaScript community, and with a focus on performance as the quality attribute of interest. All the artifacts, including the sources of the applications used in the experiments, are available online[1].

The implementation of the benchmark consists of four main phases that match the phases of the framework previously presented in Section 3. The following subsections present in detail the three first phases, while the final phase, *Result analysis*, will be presented and discussed in Section 5.

4.1. Preliminary Analysis

4.1..1. Reference Application Specification: Considering entertainment and utility applications, we leave out of the scope of our study other types of applications such as games, payment, and chat. In order to understand which features the reference application specification should support, we analyzed the most popular applications and their primary features [26], [14], [3]. This analysis is included in the benchmark website. The analysis showed that all the applications seem to somehow rely on native features such as camera access, geolocation, and notifications. A lot of them require login to access the main features of the application. Also, most applications rely on fetching content (e.g., images or videos) from an external source. Given these observations, we designed the structure presented in Figure. 3, which is representative of the mobile applications to be tested with the majority of features and the typical navigation flow.

According to this architecture, when first visiting the application is presented with a landing page, and if it is not logged in, the user can proceed to do so by navigating to the login page. On this page, after submitting the credentials, an HTTP request is sent to the backend server that will then return a unique token in case the credentials are correct, and the user

[1]https://anonymous.4open.science/r/savery

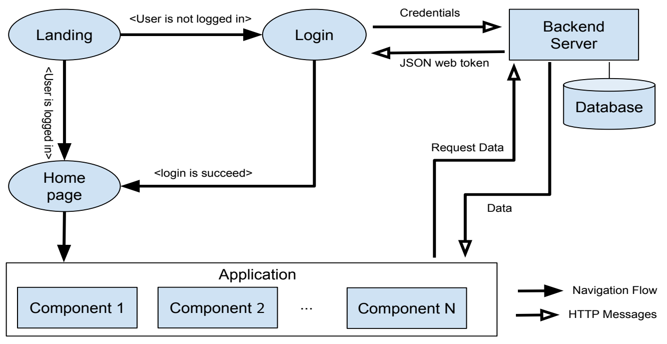

Figure 3. General architecture of the applications to be implemented.

is redirected to the homepage. On the homepage, the user can access eight carefully thought components that aim to cover the majority of the features presented in the table above.

The components are the following: **Camera**: renders live camera feed; **Geolocation**: renders user geolocation; **File Access**: when a button is clicked, a file picker is displayed to select an image from the file system; **Notifications**: after clicking a button, a local native notification is sent to the device; **Feed**: when the page mounts, several random posts are requested to our backend server; **Carousel**: when the page mounts, random images are requested and rendered to the screen, and as the user scrolls through the page, more images are requested and rendered simulating an infinite scroll; **Background Sync**: this component tracks the user connection and fetches content from the cache; **Expensive Operation**: contains several buttons to perform expensive operations in a table, such as creating 100 and 1000 rows, and swapping or deleting rows.

4.1..2. Selection of development tools: We selected seven different development tools to use in implementing applications to be tested. The following list enumerates the chosen development tools and their relevance for the benchmark: 1) **React.js**: uses a virtual DOM to build web applications, and it is, currently, the most popular development tool; 2) **Preact**: Minimalistic version of React to produce fast and simple web applications; 3) **Next.js**: Scalable and production-ready library built on top of React; 4) **Gatsby**: built on top of React to produce static web applications; 5) **Svelte**: interacts with the real DOM to build web applications; 6) **Expo**: is a framework built on top of React Native to build applications for several platforms from the same codebase; 7) **Ionic and Capacitor**: Uses web view containers to render in multiple platforms using web technologies and Capacitor to call to APIs.

4.1..3. Metrics: For the static metrics, the benchmark includes application size, line of codes, dependencies number (and dependencies number only for development), and build times. For the runtime metrics, the benchmark contains three metrics: response times, CPU usage, and RAM consumption.

CPU usage is the percentage of CPU used by each application during a certain interval [28]. An higher CPU usage will have an impact on the battery duration. **RAM consumption** is the amount of memory allocated by the application being tested, and devices can see their performance "degraded if a high percentage of the available RAM is allocated" [28]. **Response Times** are the most important metrics to mobile application users. This is the total duration that an action took to happen. Studies show that "53% of mobile users abandon sites that take over 3 seconds to load" [13].

Once measured, all the metrics presented have one thing in common: the smaller their value, the better. In Section 4-C3 we explained how each metric was collected during the Benchmark Campaign.

4.2. Preparation of the System Under Benchmark (SUB)

In the second phase, we developed the seven applications and validated them with functional testing.

4.2..1. Application Development: All the applications were developed in-house. Although this might introduce some bias in the evaluation, this will be mitigated when the project is open to the community, and engineers will be able to improve or fix the existing implementations and add implementations relative to new development tools.

To style the applications, we used a styling solution named Tailwindcss [2] for all web applications and Ionic. With this approach, we reduce the time wasted styling the applications (because all the styling is reused) and reduce the possible bugs in this process. Since Tailwindcss is not supported on React Native, we did not use any additional library to style the elements on that application.

To build the React Native application, we used Expo [3]. Expo is built on top of React Native and can produce output to the web in addition to iOS and Android platforms. Expo provides several libraries that are easier to install in an Expo project. The other applications (i.e., the applications built with Next.js, preact, and Gatsby) were built using their official command-line helpers.

For the operation, carousel, or feed component, where there was a need for adding several elements to the screen, we could have used specific libraries that virtualize the elements (i.e., only render them when they are visible on the screen) to make it more performant. However, we opt not to do it and keep the applications as simple and with fewer third-party libraries as possible. In the future, there is a possibility to add new application versions to the benchmark to compare whether these solutions impact the performance of the applications.

4.2..2. Functional Testing: To satisfy the requirements of the framework and produce a valid Testing and Measurement Tool, we found three main candidates: Appium, Selenium, and Selendroid.

We ended up choosing Appium over the others because of several reasons: i) Appium supports multiple platforms (e.g., Android, iOS) as opposed to Selenium that only supports web environments, and although we are only targeting Android in our benchmark, our framework can be easily extended to more platforms; ii) Appium can automate tests to multiple environments (e.g., web, native, and hybrid); iii) Appium allows us to write the tests with WebdriverIO, a library that has Node.js bindings (a language we are very familiar with); iv) Selendroid development ceased activity; v) Appium has a client-server architecture which allowed us to build an auxiliary Testing and Measurement Tool. Our experimental environment includes the three elements specified in the framework:

- Testing and Measurement Tool (TMT): responsible for running Appium and establishing the connection between the Android device and Appium. After the connection is established, the tool will send the automated tests to the device while collecting the metrics.Detailed in Section 4-B4.
- An Android device: connected to the TMT via USB, with the following specs: 4GB RAM, medium-end CPU (a Octa-core with 4x2.3 GHz Cortex-A72 & 4x1.8 GHz Cortex A53), GPU Mali-T880 MP4, and Android version 7.
- Remote server: The current architecture was deployed with the help of Dockerfor isolation and Nginx Proxy Manager. For each web application, an Nginx web server was used to serve the respective production build files. For the native and hybrid applications, another server was used to serve the Android APKs.

4.2..3. Workload: The benchmark workload follows the reference app specification previously defined. The main objective of the workload is to go through the application and test its components extensively without any human interaction.

Each component has a set of specific functionalities that the workload goes through with the help of automated tests, as follows. 1) Visits landing page; 2) Clicks button to request permissions; 3) Accept all the request permissions. Web applications request three permissions: notifications, geolocation, and camera. Native only requests geolocation and camera permissions because notifications are granted by default.; 4) Press the login link to be redirected to the login page; 5) Insert the correct login credentials; 6) Press the login button to be redirected to the home page; 7) Visit each component by accessing them in the navbar; 8) In each component, execute the respective functionalities; 9) After all the components are visited, open navbar and press logout

After half the components are visited, the application goes to the background to measure its CPU usage and RAM consumption while not active. A table enumerating all the performed tests is included on the benchmark website. The set of these tests altogether is our workload for both the Validation and Campaign phases.

4.2..4. Measurement and Testing Tool: Since our objective is to make this study an open-source project, we implemented this tool using TypeScript[4], a superset of JavaScript, designed to be more scalable and easier to maintain.

To create the workload above-mentioned, we used automated tests with the help of multiple testing frameworks of the JavaScript environment: WebdriverIO[5], Mocha[6] and Chai[7].

[2]https://tailwindcss.com/tailwindcss

[3]https://expo.io/expo.io

[4]https://www.typescriptlang.org/

[5]https://webdriver.io/

[6]https://mochajs.org/

[7]https://www.chaijs.com/

Appium[8], an automation testing framework already mentioned before, is then responsible for sending the tests and interacting with the device.

In the campaign phase, while the tests are running, this tool also sends ADB (Android Debug Bridge) commands periodically to the connected Android device to collect the CPU usage and RAM consumption. All these measures and test results are then saved into JSON and Sqlite files for further analysis. While the automation tests were sent to the target device in the main thread, we used a library called Threads.js[9] to spawn a new thread to send the ADB commands referred to in Section 4-C3 to collect the metrics every 200 ms. This tool needs to be running on a device with a valid Android SDK installation to be compatible with Appium.

After the applications were implemented, the Testing and Measurement Tool injected the workload to validate the applications. Some fixes were needed, but once all the applications were validated, they were ready to proceed to the next phase.

4.3. Benchmark Campaign

In the third phase, also with the help of the measurement tool explained in the previous section, the same workload was injected again but this time while gathering the previously defined metrics. We will start by explaining how each metric was collected and then how the campaign was conducted.

4.3..1. Configurations: Before starting each execution, the campaign must comply with the following configurations:

- All the web applications and the backend server must be served with HTTPS (also a requirement for the validation phase).
- Applications must be isolated from each other in the remote server to have fair comparisons.
- Permissions are not granted to any application to emulate what happens in the real world.
- The target device that is running each application cannot be running any other applications. We made sure the CPU and RAM overall utilization were always below 3% for 5 straight seconds before starting each campaign.

4.3..2. Execution of SUBs: As for the benchmark campaign flow, we follow the suggestion from our framework mentioned in Section 3-C, with the particularity that n our benchmark, we repeat the campaign at least 6 times for each application.

4.3..3. Measuring metrics: To measure the **size** of web applications, we used the service $lighthouse - metrics.com$ and $lighthouse - metrics.com$, which provides the total transferred resources size. For native and hybrid applications, this value is just the size of the final Android Application Pack (APK). The `lines of code` were measured with the help of cloc[10], a great tool to count physical lines of source code in many programming languages.The `dependencies number` was calculated by analyzing the `package.json` file. To compute the `build time`, we used the `time`

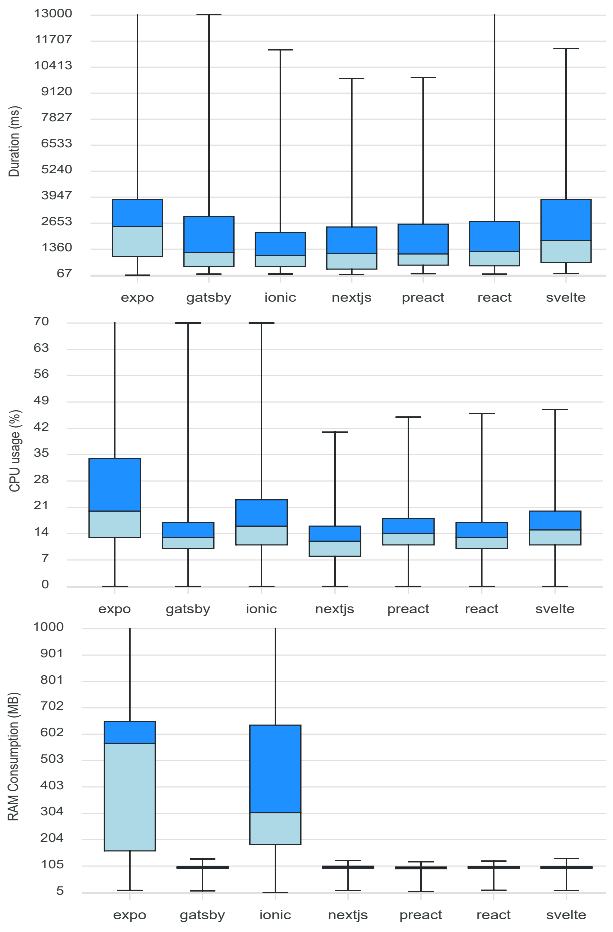

Figure 4. Results per application: Average Test Duration (ms), CPU Usage (%) and RAM Consumption (MB).

command from Unix. For native and hybrid applications, an additional step was needed. Since these applications need to be built using Android Studio, this duration also needs to be accounted for. Considering Expo uses its cloud-based system to build the APKs, the duration did come from this system instead of Android Studio.

Runtime metrics are trickier to collect, and to respect the framework requirements, we developed a solution that was non-intrusive and had minimal impact on the benchmark.

To collect the `CPU usage`, our Measurement and Testing Tool sends the `adb top` command every 200 ms during the execution of the tests via USB on the Android device. After analyzing each test's duration, we concluded that 200ms would be small enough to intersect each test and provide the CPU usage during the execution of each test.For web applications, the package monitored through the above command is the browser on which the applications are running (i.e., Google Chrome in this study, hence the package name used in the above command is `com.android.chrome`). In the case of native or hybrid applications, the package considered is the respective APK's package name. The process of collecting the `RAM consumption` is similar to the previous one with the difference of the ADB command: `adb dumpsys meminfo`

[8]http://appium.io/
[9]https://threads.js.org/threads.js
[10]https://github.com/AlDanial/cloc

command was used instead.

Finally, the `response times`, in our study, correspond to the test duration in ms. We opt to not include code to collect these values directly in the applications' source code because, as said before, we treated all the applications as a black box. Therefore, we use the duration of each test (i.e., the time taken by each application to complete each function) to compare the different applications. This value was measured using our Measurement and Testing Tool with the help of the `perf_hooks` module from Node.js.In all the tests, we tick the instant immediately before the test starts and immediately after it ends. The subtraction between these two values provided the test duration.

5. Experimental Results

The main objectives of the experiment were to demonstrate the applicability of the approach and the benchmarking framework proposed and also to analyze and compare the different mobile development tools (or frameworks). It is worth mentioning that we repeated all experiments 6 times to observe the variability of the results and the presented results correspond to the average values of all 6 executions. The complete data and results are available online[11].

5.1. Overall results

Figure. 4 presents a brief summary of our findings. Starting with the average duration of the experiments, surprisingly Ionic, the hybrid app, presents the best results overall with Next.js and preact as close competitors. Next.js and preact achieving low response times was expected since both development tools are built with performance as the main requirement. Expo (React Native) presents the worst results, which were unexpected since the Expo uses Native components.

Overall, regarding CPU and Memory consumption, Expo and Ionic are the ones with the highest consumption. This was also foreseen because they are not fully native. Although this benchmark did not include a completely native application (e.g., produced using Kotlin) we expect that it would have better results (less RAM and CPU consumption and a lower test duration) than both Ionic and Expo. The average values also show that the web applications behave similarly in terms of CPU and RAM usage. Svelte consuming slightly more CPU than the React-based competitors was not a surprise as interacting with the Real DOM is an expensive operation. Although Svelte's average test duration was slightly higher than the other web applications, our findings show that the Virtual DOM used by React.js or Vue may be unnecessary overhead for some use cases [10]. Fast applications (i.e., applications with smaller test duration) can be achieved without a DOM abstraction such as the one used by React.js or Vue development tools. Moreover, our findings also show that the React-based development tools got similar results.

[11]https://savery.vercel.app/

5.2. Results for Test Duration

Figure. 5 presents the average duration per test for each application.

In the first test, `t-landing-1`, we wait until the landing page is presented after the application starts. By analyzing the results, Expo is the application that takes more time to start up. On the other hand, as expected, preact aces it by taking less time to complete the same task. In the second test, where they accept the permissions needed for the tests, the web applications behave similarly. Since the hybrid and native applications only need to accept two permissions take less time (explained in Section 4-B3). Toggling the navbar (represented in tests `t-home-4` and `t-home-5`) presents similar results in all applications but Expo. This was expected since the Expo adds by default an animation that delays the navbar appearance or removal.

Surprisingly, despite the same code being used in all React-based applications, the Gatsby application failed to realize, and the device went offline and timed out in this test (`t-bgSync-2`). The other applications behave similarly. To retrieve an image from the cache, in test `t-bgSync-4`, all the web applications performed similarly (both native and hybrid). Since the web applications react fast to lack of internet and can quickly retrieve content from the cache means that **Web applications can already provide offline functionalities**.

In the tests where native features such as Geolocation, Camera, or accessing the Filesystem were used, web applications behave similarly and better than the native application Expo but overall worse than the Ionic.

Regarding loading content from outside sources, web applications take less time. We highlight the test `t-feed-1` where Next.js excels due to Incremental Static Regeneration. This feature provided by Next.js allows the users to be always presented with instant content (it is more detailed in [21]). Conversely, the other applications took more than two seconds to show the actual data.

Finally, we highlight the final tests that perform operations on a table data structure. Expo crashes in the creation of 1000 rows. A Github issue[12] confirmed our assumption that rendering a large number of elements can result in sluggishness and even crash a React Native application. Therefore, this use case needs to be accounted for and implemented carefully. React Native team published resources on how to solve this problem[13]. For improving the performance in React-based solutions react-virtualized [27] can be used.

5.3. Results for CPU Usage

Overall, native and hybrid applications consume more CPU than other applications. Figure. 6 presents the CPU usage comparison between Expo, Ionic, and preact, three different types of applications (i.e., web, hybrid, and native respectively). Expo uses less CPU in all tests with some minimal exceptions that are not worth mentioning. Between Expo and Ionic, the

[12]https://github.com/facebook/react-native/issues/13413

[13]https://reactnative.dev/docs/optimizing-flatlist-configuration

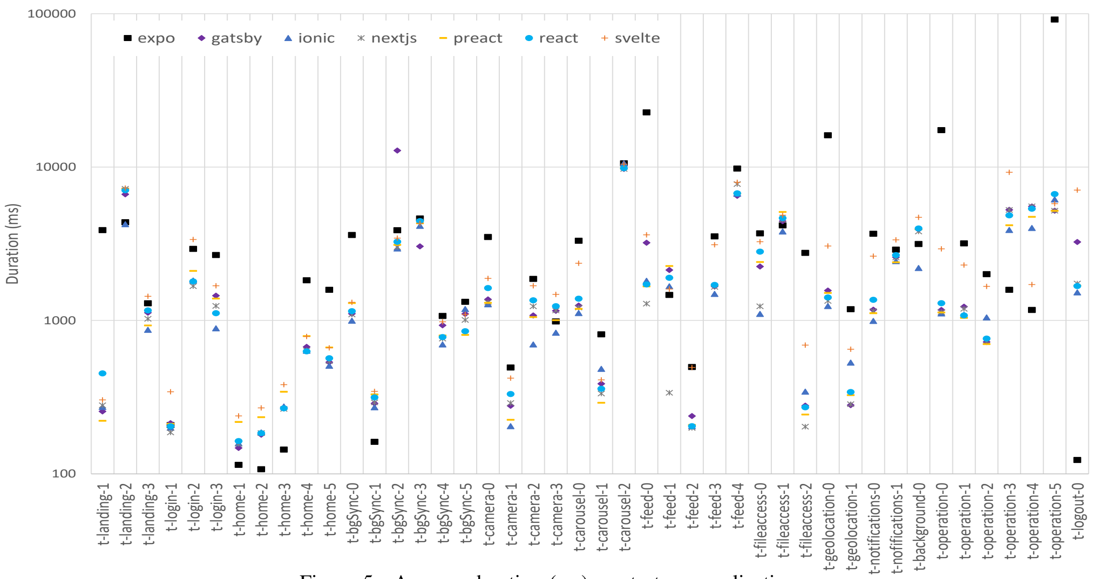

Figure 5. Average duration (ms) per test per application.

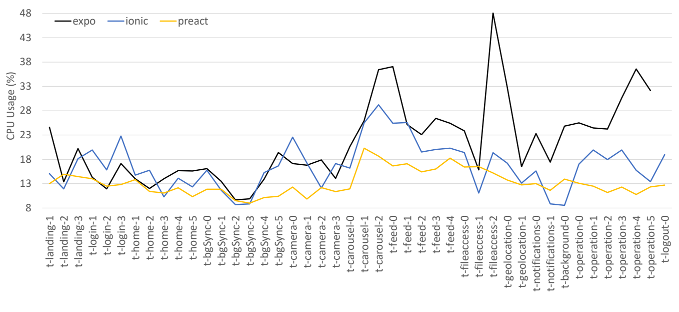

Figure 6. CPU Usage (%) per test between Expo, Ionic, and preact.

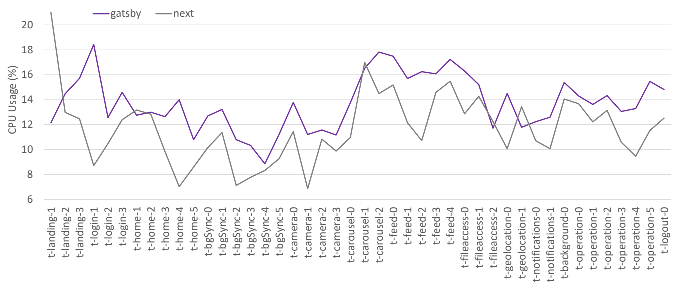

Figure 7. CPU Usage (%) of Gatsby and Next.js per test.

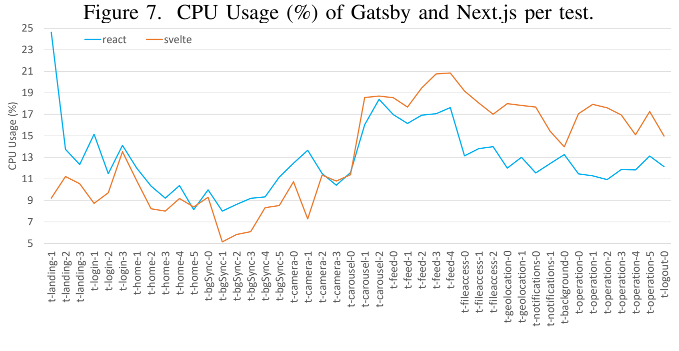

Figure 8. CPU Usage (%) of React and Svelte per test.

results are similar until the point where large amounts of elements are added to the screen. From this point, Expo starts using more CPU than Ionic. If the device on which these types of applications are running does not have good specifications, web-based alternatives prove to be an efficient solution.

Figure. 7 presents the comparison between Gatsby and Next.js, two direct competitors to build web applications. Overall, the CPU usage of the Next.js application is always slightly lower. This is expected since both applications are React-based and use the virtual DOM. At the beginning of the test, the Next.js application is consuming more CPU. It might be because it bundles more assets than Gatsby (observed in application size in Section 5-E). However, in the second test, this value is already lower than the competitor's value and remains lower during all tests except for two of them. Next.js proves to be an efficient solution in terms of CPU usage as well.

Figure. 8 presents the comparison between Svelte and React.js. The first one is a very promising tool that uses the real DOM against the current standard to produce web applications and the second one uses the virtual DOM. Similar to the previous comparison, React.js starts with a higher CPU usage due to the fact that it bundles more assets than Svelte and the JavaScript code contained in these assets needs to be parsed by the browser. Then, React.js reduces the usage but stays slightly above the Svelte application. This only changes when the more expensive operations start (i.e., scrolling in Carousel).This heavier usage of the CPU on the Svelte application proves that for applications requiring a higher number of interactions with the DOM while fetching content from the internet, interacting with the real DOM might be inefficient (refer to [10] for more details). A similar situation happens with the operation component tests.

5.4. Results for Memory Consumption

Figure. 9 presents all the RAM measures during one execution (out of 6) for each application. We compared this same figure in 6 different runs and they are all similar. Again, the hybrid and native applications are the ones with the highest consumption. On the other hand, web applications consume similar values compared to each other.

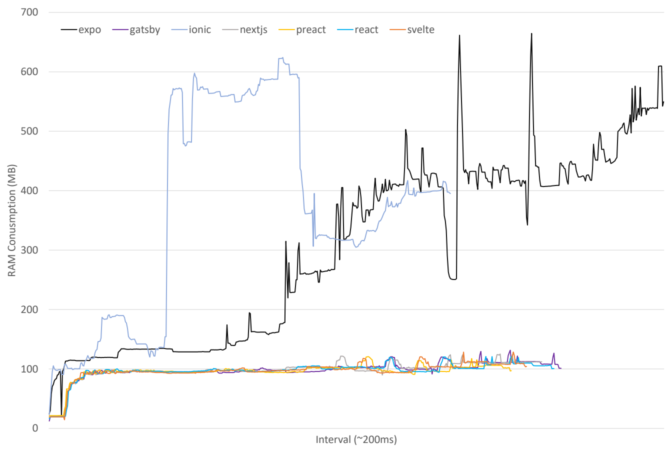

Figure 9. RAM Consumption (MB) per application (ticks every 200 ms).

It is also observed that from one point, the RAM allocated by Expo starts inflating and is only released when the application crashes in the last operation component tests. This happened because of its implementation: since we did not use any performance optimizations for the other applications, to have a fair comparison we opt to not implement any performance optimizations for rendering lists in Expo. The Expo application becomes sluggish and impossible to use after rendering several elements first in the Carousel component, then in Feed and Operation components. Although it appears that Expo releases the memory in the last tests, this is not true, the application ends up crashing in the last Operation component when rendering 1000 rows to the screen. On the other hand, even though Ionic allocates a huge amount of memory for some tests it seems to perform efficiently and releases this memory when not needed.

5.5. Results for Static Metrics

Table I presents our findings after analyzing the static metrics data. Although this was not the primary focus of our benchmark, we opt to collect this type of metrics and discuss them to prove that our framework is extensible not only to gather execution metrics but also metrics related to the source code and the development process.

TABLE I
STATIC METRICS FOR EACH APPLICATION.

App name	Size (KB)	LoC	Total number of dependencies (dev)	Avg. build time (s)
Expo	52892.0	1359	29 (3)	198.00
Gatsby	211.0	1564	11 (6)	39.38
Ionic	5301.0	1737	38 (12)	61.53
Next.js	243.0	1658	9 (5)	28.12
Preact	60.5	1657	13 (8)	85.33
React.js	190.0	1610	13 (7)	19.71
Svelte	54.2	1458	13 (7)	23.30

Regarding the application size, Svelte presented the smallest value (around 54.2 KB). Since Svelte compiles everything at build time, it does not need to bundle the whole library code to make computations at runtime as React and many other development tools do [10]. Consequently, Svelte application size for small applications is lower than a react-equivalent application.preact has the smallest size of all the React-based applications since preact is a minimalistic version of React.js. The Ionic application size is 5MB which is great for a Native Android application. On the other hand, Expo is 10x larger. Since we were using Expo instead of plain React Native, to build the application, the Expo Cloud system was mandatory to build the application. However, this can be avoided by ejecting the Expo application. After the application is ejected, the APK can be built locally, and perform certain optimizations to reduce the application size (also, the build times will be lower than with the expo cloud system).
Regarding the dependencies number, Ionic required a high number to transform a valid React.js app (13 dependencies) into a valid hybrid app (38 dependencies).
Svelte, which presented better Build Time results than Next.js but is inferior to Craco, uses a bundler tool called Rollup. The comparison between the different JavaScript bundler tools is out of the scope but a great comparison is provided in [16].

6. Threats to Validity

In this section, we discuss the points that threaten the validity of our approach and obtained results.

- **Representativeness of the Reference Application Specification**: A major flaw in several studies is the lack of representativeness of the applications under assessment. To mitigate this issue, we analyzed the most popular applications and their features. Then, we built a Reference Application Specification that covers the majority of the use cases observed.
- **Applications developed in-house**: Although this might introduce some bias in the assessment, this will be mitigated once the project is open to the community. Developers will then be able to improve or fix the existing implementations and add new development tools.
- **Not including application completely native**: Our benchmark included React Native but through a cross-platform tool. A complete native application is not included due to time constraints, but this does not preclude the future inclusion of the fully native app. Also, it does not affect the observations on the evaluated tools.

7. Conclusion and Future Work

This paper proposes a new framework for assessing different mobile application development tools considering several properties, such as performance, reliability, or security. To demonstrate our framework, we conducted a performance benchmark and compared seven popular development tools. The results show that our framework can indeed be used to compare different development tools. Our findings also showed that Ionic was heavy Memory and CPU presented the fastest results. Also, tools that claimed to be built with performance as first requirement (e.g., Next.js and preact) were the fastest among the web applications tested.
As future work, we are planning to disseminate our study through the community by open-sourcing the project. Also,

we plan to perform more rounds of the benchmark including more development tools (and improving the existing ones) in different Android devices. In addition, a more ambitious goal for the future is to support iOS development tools.

Acknowledgment

This work is funded by the FCT – Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit – UIDB/00326/2020 or project code UIDP/00326/2020; and by Project "NEXUS Pacto de Inovação – Transição Verde e Digital para Transportes, Logística e Mobilidade". ref. No. 7113, supported by the Recovery and Resilience Plan (PRR) and by the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022.PC645112083-00000059 (project 53), Component 5 - Capitalization and Business Innovation - Mobilizing Agendas for Business Innovation.

References

[1] Andreas Biorn-Hansen, Tim A. Majchrzak, and Tor-Morten Gronli. Progressive Web Apps: The Possible Web-native Unifier for Mobile Development:. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies*, pages 344–351, Porto, Portugal, 2017. SCITEPRESS - Science and Technology Publications.

[2] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 37–48. USENIX Association, 2011.

[3] Julia Chan. Top Apps Worldwide for January 2021 by Downloads.

[4] Lukas Dagne. Flutter for cross-platform App and SDK development. page 37.

[5] David Fortunato and Jorge Bernardino. Progressive web apps: An alternative to the native mobile apps. In *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2018.

[6] Rebecca Fransson, Alexandre Driaguine, and Johan Hagelbäck. Comparing Progressive Web Applications with Native Android Applications. page 59, 2017.

[7] Rasmus Fredrikson. Emulating a Native Mobile Experience with Cross-platform Applications. page 19.

[8] Abhi Gambhir and Gaurav Raj. Analysis of Cache in Service Worker and Performance Scoring of Progressive Web Application. In *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pages 294–299, Paris, June 2018. IEEE.

[9] Michael Gonsalves. *Evaluating the mobile development frameworks Apache Cordova and Flutter and their impact on the development process and application characteristics*. Thesis, June 2019.

[10] Rich Harris. Virtual DOM is pure overhead, January 2021.

[11] Kelsey L. Johnson and Mark M. Misic. Benchmarking: a tool for Web site evaluation and improvement. *Internet Research*, 9(5):383–392, December 1999.

[12] Karama Kanoun and Lisa Spainhower. *Dependability benchmarking for computer systems*, volume 72. Wiley Online Library, 2008.

[13] David Kirkpatrick. Google: 53% of mobile users abandon sites that take over 3 seconds to load.

[14] John Koetsier. Here Are The 10 Most Downloaded Apps Of 2020.

[15] krausest. krausest/js-framework-benchmark: A comparison of the perfomance of a few popular javascript frameworks, October 2020.

[16] Sonja Laurila. Comparison of JavaScript Bundlers. page 55.

[17] Kim Maida. How to Manage JavaScript Fatigue, December 2020.

[18] Ivano Malavolta. Beyond native apps: web technologies to the rescue! (keynote). In *Proceedings of the 1st International Workshop on Mobile Development - Mobile! 2016*, pages 1–2. ACM Press, 2016.

[19] Ivano Malavolta, Giuseppe Procaccianti, Paul Noorland, and Petar Vukmirovic. Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 35–45. IEEE, May 2017.

[20] mathieuancelin. mathieuancelin/js-repaint-perfs: Playground to test repaint rates of JS libs, October 2020.

[21] Lee Robinson. A Complete Guide To Incremental Static Regeneration (ISR) With Next.js.

[22] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. *In Proc. of CCA*, 8:228, 2008.

[23] Farid Said Tahirshah. Comparison between Progressive Web App and Regular Web App. page 69.

[24] TechEmpower. TechEmpower/FrameworkBenchmarks: Source for the TechEmpower Framework Benchmarks project, October 2020.

[25] the benchmarker. the-benchmarker/web-frameworks: Which is the fastest web framework?, October 2020.

[26] Lionel Valdellon. 60 Most Popular Apps on the App Store and Google Play.

[27] Brian Vaughn. bvaughn/react-virtualized, May 2021. original-date: 2015-11-03T00:48:07Z.

[28] Michiel Willocx, Jan Vossaert, and Vincent Naessens. Comparing performance parameters of mobile app development strategies. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 38–47, 2016.

[29] Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad Van Moorsel. Resilience assessment and evaluation of computing systems. 2012.