IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 19, 2022, accepted May 7, 2022, date of publication May 13, 2022, date of current version May 23, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3174855

On the Performance of Cloud-Based mHealth Applications: A Methodology on Measuring Service Response Time and a Case Study

DEVASENA INUPAKUTIKA[1], (Graduate Student Member, IEEE), GERSON RODRIGUEZ[2], DAVID AKOPIAN[1], (Senior Member, IEEE), PALDEN LAMA[3], PATRICIA CHALELA[4], AND AMELIE G. RAMIREZ[4]

[1]Electrical and Computer Engineering Department, University of Texas at San Antonio, San Antonio, TX 78249, USA
[2]Google, Mountain View, CA 94043, USA
[3]Computer Science Department, University of Texas at San Antonio, San Antonio, TX 78249, USA
[4]University of Texas Health Science Center, San Antonio, TX 78229, USA

Corresponding author: Devasena Inupakutika (devasena.inupakutika@gmail.com)

This work was supported in part by the Susan G. Komen under Award SAB160005, in part by the Mays Cancer Center under Grant P30 CA054174, and in part by the Redes En Acción under Grant U54 CA153511.

ABSTRACT With the increasing use of smartphones, performance monitoring and the analysis of mobile applications (apps) are gaining momentum. Smartphones are resource-constrained devices. Thus, mobile apps typically rely on cloud services for the execution of resource-intensive functionalities, storage, and computation power. Measuring the user experience is crucial for the development and maintenance of mobile apps. Such characterization requires testing specific traits such as network connectivity, battery levels, server loads, and operating conditions. This paper presents a technique for the measurement-based performance assessment of cloud backend and mobile networks that support mobile app services. The feasibility of the technique is demonstrated through a representative case study of an app developed for medication adherence management among breast cancer patients undergoing endocrine hormone therapy (EHT). The app leverages cloud technologies to provide a portable, cost-effective, and convenient monitoring environment. Nonfunctional performance and load testing is performed by modeling third-party cloud backend services. The experimental results of the case study demonstrate the feasibility of the approach for monitoring and analyzing the backend service response times with different mobile device configurations, such as regular or power-saving battery modes and LTE or Wi-Fi mobile network connectivity, under server loading. The methodology is validated through statistical analyses of the experimental performance data involving confidence intervals, tail latencies, and analysis of variance. The results address the occurrence of server loading and its impact on the response times which relates to the quality of the user experience. We establish the effect of server loading on the responsiveness of the user interface (UI) of the mobile app considered in this case study. The proposed technique will allow developers to conduct similar measurement-based performance studies for various mobile apps leveraging cloud-based backend services.

INDEX TERMS Application behavior, cloud databases, healthcare, mobile applications, performance measures, testing.

I. INTRODUCTION

In recent years, there has been an exponential proliferation of mobile devices and an increase in the transition to mobile applications (apps). The number of smartphone users worldwide is expected to reach 7.074 billion by 2024 [1]. Mobile devices facilitate ubiquitous responsive services such as push-notifications, and messaging, along with rich audiovisual content. Mobile apps elevate their competitiveness by facilitating economic resource consumption and the provision of key functionalities to users. Most mobile apps for everyday use (e.g., Fitbit, and Uber) depend on network connectivity, external web services, and sensors, which continuously interact with each other. The rapid

The associate editor coordinating the review of this manuscript and approving it for publication was Md. Abdur Razzaque.

growth, feature richness, and augmented network traffic of such apps creates considerable overhead in terms of the consumption of device resources consumption—including power, CPU, and bandwidth—as well as the mobile network resources. These issues have attracted attention from both mobile app developers and full-stack developers who aspire to comprehend mobile apps' resource usage and its effects on interaction with a backend database. Here, challenges regarding software correctness also arise. In addition to functional accuracy, response time and resource consumption are becoming critical specifications for ensuring a better user experience. Context awareness, mobile device fragmentation, event-driven programming paradigms, and constantly evolving mobile technologies and frameworks have added further challenges. Given the time and effort required to consolidate different possible test scenarios and generate relevant test cases, automated testing [2] has become crucial for performance assessment and the delivery of high-quality apps.

Automated testing tools help validate both functional and non-functional requirements [30], [31]. Functional testing verifies the operations and actions of an app. Test automation in functional testing further helps to automate test cases and features that are constantly regressing. Non-functional testing also known as quality of service (QoS) testing, verifies the behavior of an app and checks its non-functional parameters with the aim of improving the app's usefulness, portability, efficacy, maintainability, and collection of metric data for research and app performance analysis. Crucial non-functional parameters include performance, loading capacity, usability, efficiency, reliability, stability, portability, and security.

Performance testing of mobile applications addresses two primary aspects: 1) the performance of the services or servers to which the app is connected [32] and 2) the performance of the mobile app client on the host device [33]. Testing of the former focuses on the application of load on the servers that host mobile apps (including typical web servers and serverless database backend-as-a-service). This type of testing is very common, and a number of existing tools have been adapted to conduct performance studies of web mobile apps (for example, LoadStorm,[1] Apache JMeter,[2] and NeoLoad[3]). However, it is not easy to adapt for a developer who either does not own a web server or leverages a serverless architecture for mobile app development. The latter type of testing focuses mostly on assessing the host platform resource consumption of a mobile app. For example, in the case of a medical reference mobile health (mHealth) app, this type of testing will involve executing the ''medical educational reference content access'' feature and evaluating the corresponding memory consumption of the mobile app. Such testing predominantly involves validating the app's performance through platform-specific performance tests [34] [35]. In this work, we present a methodology for conducting performance testing addressing both aforementioned primary aspects to study the performance of mobile apps in the context of server loading and the effect on the responsiveness of the UI based on response time.

This paper analyzes the non-functional parameters such as the mobile network and battery modes affecting a mobile app's responsiveness and performance on Android and iOS platforms under load on a cloud-based backend database (DB) server. In general, many different performance metrics exist that reflect non-functional requirements. In this paper, we focus on the server response time (T_{response}) because a large number of cloud applications and services are delay sensitive, such as healthcare, file hosting, mobile cloud sensing, and location-based mobile services. Thus, response time is an important performance metric for the end-user experience. To validate our technique, we first developed a bilingual (English and Spanish), cross-platform hormone therapy (HT) Patient Helper app that exploits the rich existing resources and development tools offered by Google and Apple. The app is designed for breast cancer patients undergoing endocrine HT (EHT) [10]. Google Firebase backend services are used to facilitate cloud storage and to send notifications (Firebase Cloud Messaging[4] (FCM)) to mobile devices through a cloud manager. We further conducted a performance analysis of this case study app from the perspective of the smartphone's battery modes to investigate resource usage under different mobile network connectivity settings (Long Term Evolution (LTE) vs. Wi-Fi) and its effect on the server response times for service requests from a real-time cloud DB with varying loads.

The main contributions of this paper are summarized as follows:

- First, we propose a methodology for measuring cloud DB or server response times as a metric for non-functional performance and performing load testing of mobile apps that leverage cloud technology by modeling third-party cloud backend services.
- We report conduct a comprehensive study of the effect of server loading on the responsiveness of a mobile app's user interface (UI), which influences the end-user quality of experience (QoE), with the proposed methodology for a case study mHealth app.
- We study and present a statistical analysis of the response times under different configurations in terms of mobile network connectivity and battery modes in the presence of server loading.

Additionally, for ease of reading, Table 1 presents a list of abbreviations that are used in the following discussion. The structure of this paper is as follows. Section II presents background information and related work on mobile apps in general as well as on the use of cloud DBs

[1] https://loadstorm.com/
[2] http://jmeter.apache.org/
[3] https://www.neotys.com/neoload/overview
[4] https://firebase.google.com/docs/cloud-messaging

TABLE 1. Comprehensive list of abbreviations for ease of reading.

Abbreviation	Description
app	Application
API	Application programming interface
ARC	Automatic reference counting
AUT	Application under test
DB	Database
EHT	Endocrine hormone therapy
FCM	Firebase Cloud Messaging
GC	Garbage collection
GUI	Graphical user interface
JSON	JavaScript Object Notation
LTE	Long Term Evolution
MAX_ITERATIONS	Number of cycles per spawn
MAX_SPAWN	Number of simultaneous applications
mHealth	Mobile health
NUM_REQ_PER_THREAD	Number of requests per thread
NUM_THREADS	Number of threads/ simultaneous users per spawn
OS	Operating system
PS	Power saving
QoE	Quality of experience
QoS	Quality of service
UI	User interface
CI	Confidence interval
sdv	Standard deviation
sz	Sample size
ANOVA	Analysis of variance

as a service and introduces state-of-the-art performance analyses utilizing cloud technology. Section III describes the design of the proposed methodology for performance testing. Section IV describes the case study app, with a brief justification of the design choices and structure of the project. Section V discusses the application of the proposed testing methodology to assess the performance of our developed case study prototype mHealth HT Patient Helper app and summarizes the dataset representing our user scenario. Section VI presents the experimental results with the statistical analyses of the tests conducted under different battery and network modes and discusses the threats to the validity of the proposed approach. Section VII presents a discussion and lessons learned. Finally, Section VIII presents concluding remarks and suggests directions for future work.

II. BACKGROUND AND RELATED WORKS

A. UI AUTOMATION FRAMEWORKS AND RELATED WORKS

Mobile apps are susceptible to performance issues. The system may cause delayed loading of emergency app content, stop the execution of an app, or throw a warning on the app UI that the ''Application is not responding''. Several contributions highlighting issues related to the performance of Android apps, such as poor responsiveness [36]–[38], have been presented in the literature. Furthermore, recent work [39] on a systematic literature review of the automated testing of Android apps offers a detailed analysis of the key aspects of testing mobile apps and provides a taxonomy for clearly summarizing test objectives (e.g., performance and energy), test targets, levels, and testing methods, with corresponding frameworks in each category (e.g., model-based, search-based, random, A/B, fuzzing, and mutation). Performance diagnosis often requires automatic execution of the app under test (AUT). Hence, UI automation frameworks are the most suitable option for testing mobile apps and play an important role in performance assessment. They are used for replicating user interactions and are used as part of the overall experimental setup for system performance testing. In particular, script-based testing is the most widely used approach (e.g., UI Automator, MonkeyRunner, and Robotium). Event sequences may also be recorded during the manual operation of an app to generate replayable scripts using record-and-replay tools (e.g., MobiPlay, Reran, and SPAG-C). As an complement to these semiautomated approaches, fuzz testing approaches (e.g., Monkey and Dyn-odroid) generate random input sequences to exercise Android apps. Model-based (MB) approaches (e.g., AndroidRipper, SwiftHand, and PBGT) focus on generating a finite state machine model and event sequences to traverse the model. Other works have also used Calabash (suitable for both Android and iOS) [40] and graphical user interface (GUI) ripping methods for regression tests to exercise the target AUT.

The design and development of automated test suites is a complicated task that should enable the examination of an app's smartphone resource usage and context in terms of the network connectivity type, device and operating system (OS) environment. Both the network resources and the function calls of a mobile app both directly and indirectly impact its usage of smartphone resources. These aspects are critical for the testing and, consequently, QoE improvement of mobile apps. Even for mobile apps, software testing is one of the most widely used quality assurance techniques. It is thus important to develop a relevant set of tools and methodologies for automated testing that focus on an app's GUI, control flow, overall functionality and performance-specific non-functional aspects. The study presented in [4] discussed the following aspects of automated mobile app testing: 1) testing frameworks for structuring the testing process and 2) GUI ripping tools for simulating real user events, which fundamentally differ in the exploration strategies (including reverse engineering approaches) used for building test suites [5]–[8]. Regarding testing frameworks, [3] extensively surveyed state-of-the-art tools and services for supporting mobile app testing processes from both the academic and industrial research perspectives. The focus was on 1) automation frameworks/application programming interfaces (APIs), 2) record-and-replay tools, 3) automated test GUI input generation techniques, 4) bug and error reporting/monitoring tools, and 5) mobile testing services, including cloud-based and device streaming tools. Additionally, the study in [3] proposed a conceptual, fully automated system architecture for mobile app testing based on the continuous, evolutionary,

and large-scale (CEL) principles. It was suggested that automated testing tools for mobile apps should address several development restrictions: 1) the restricted budget/time available for testing, 2) the need for various types of testing (e.g., UI responsiveness based on response times), and 3) the pressure from stakeholders and users for continuous delivery. In recent research, a novel strategy has also been implemented to explore app behavior using a static–dynamic approach to UI model generation for MB testing (MBT) called AMOGA [9] to understand mobile- or platform-specific faults. Nevertheless, despite the availability of such a comprehensive set of methods, techniques, and tools for automated testing, none is suitable by itself for the assessment of study of mobile app performance for several reasons, including organizational and personal preferences, device fragmentation, and the absence of a mobile-specific set of conditions when a particular test case fails. To the best of our knowledge, no single model to date can integrate all of the various aspects of mobile app operation, such as OS (Android or iOS), context, GUI, usage, and domain. Given that the choice of a testing framework depends on specific testing goals and research contexts, we devise a performance methodology that focuses on the performance-based non-functional aspects of testing and addresses tests to observe the effect of server loading on UI responsiveness under various mobile network and battery conditions based on response times.

B. CLOUD DATABASE-AS-A-SERVICE

For researchers and practitioners, cloud computing is emerging as an alternative to grids, clusters, and production environments [11]. As a service paradigm [12], cloud DBs that can support multiple mobile apps based on the internet are of great interest thanks to the advantages of cloud computing—such as resource elasticity, pay-as-you-go cost models, and scalability when deploying data-intensive apps—which make it possible to achieve higher throughput with computing resources, such as DB servers, in cases of increased workload. [13] provided various options for deploying the DB tier of software apps on cloud platforms. Hence, the cloud is considered a viable alternative to on-premises computing that ensures data safety and is easily accessible on an as-needed basis [14]. However, several concerns should be considered in terms of cloud DB performance with respect to data retrieval and access. Several studies have focused on a comparative analysis of cloud and traditional DBs from the architectural perspective [15]–[17]. Reference [18] presented a quantitative analysis comparing the performance of on-premises and Azure SQL Server DBs. In this paper, we focus on the T_{response} of the cloud DB as a metric of analysis, as in [18]. We leverage Google Firebase as a cloud-hosted NoSQL DB for the current study. Additionally, we report the statistical analyses including the 95^{th}-percentile confidence intervals for the response times of the requests.

C. EFFECTS OF MOBILE APPS' PERFORMANCE CHALLENGES ON HEALTHCARE

With the rapid growth of mHealth apps, an increased need has arisen for app development processes to include an assessment of the app's performance. Previous researchers [19]–[21] and mobile app designers have made significant efforts to present developers with best practices and guidelines for improving the performance of mobile apps. However, they have not investigated the cloud-based performance aspect. Even so, there is still a noticeable gap between research and practical deployment in terms of addressing performance bottlenecks by developers. References [20] and [24] analyzed Android and iOS apps, respectively, and found that unresponsiveness and significant resource usage were the main causes of negative user reviews. These factors are prime concerns that cannot be compromised in mHealth apps that require users and patients to keep track of their symptoms or issue reminders to take medication. To reach the true potential of mHealth in presenting unparalleled opportunities to increase patient engagement, reduce healthcare costs, and improve outcomes [22], [23], proper validation of the technical performance of the relevant apps is needed. Furthermore, healthcare services are highly time sensitive and require QoS guarantees in terms of important parameters such as delay. Therefore, the quantitative measurement of such parameters within an mHealth system is useful.

Thus, we study the performance of a case study HT Patient Helper app by identifying the factors affecting app–cloud communication. Using both Android and iOS versions of the app, we analyze the impact of the device's battery conditions (full battery vs. power-saving mode) and network connectivity (LTE cellular network vs. 802.11 Wi-Fi) on response time with varying loads on the Firebase DB.

III. DESIGN OF THE PERFORMANCE STUDY

In this section, we present our devised methodology for studying mobile app performance based on our testing goals outlined in Section I. We model third-party cloud backend services to examine the performance in a representative testing scenario for end users accessing the app. The approach is developed around a testbed that allows us to run a specific test scenario. The testbed is used to measure the performance implications of the test scenario under different server or DB workloads. We conducted a study in which the server response time (measured in milliseconds (ms)) was our dependent variable, while the test scenario and DB loading parameters (discussed in Section III.B) were the independent variables.

A. PERFORMANCE MODULE

This section describes the experimental setup used to automatically measure the service response times of mobile apps. We considered that the AUT would be run in a controlled environment. We instrumented a testbed in which

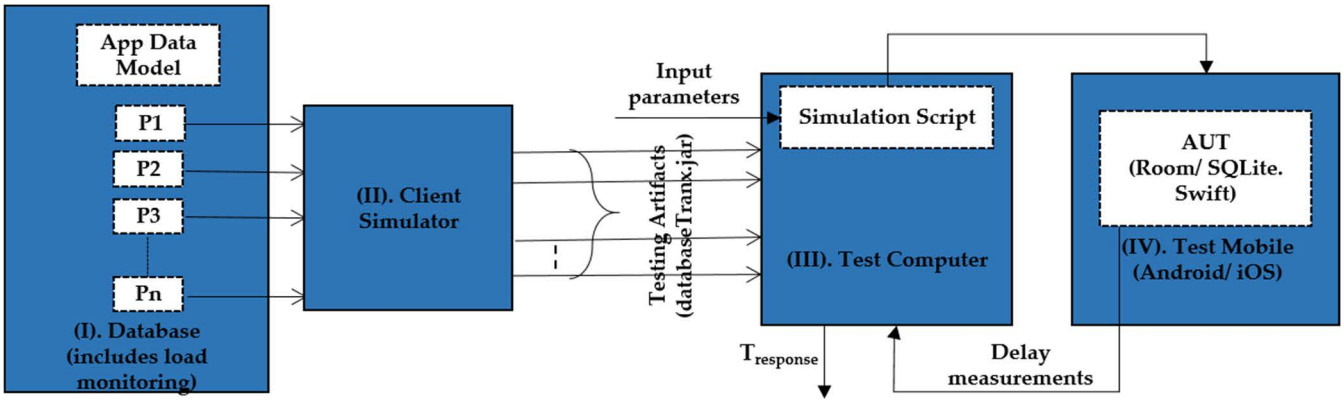

FIGURE 1. Testbed and workflow of the performance module.

multiple test users were simulated running the AUT. The complete *performance module* generated a workload — that is, it loaded the entire server or DB based on a set of passed input parameters (independent variables). The testbed simultaneously collected the output response measurements in an SQLite DB for further analysis of the complete execution. We retained complete control of the AUT and the DB, and all tests were repeatable and automatically performed. The complete *performance module* and testing methodology were run by varying the input parameters specific to the DB load and the testing campaign, the details of which are discussed in Sections III.C, IV and V. The recorded measurements from all experiments were then post-processed and analyzed. The results are further discussed in Section VI.

Fig. 1 depicts our complete testbed. The core component is the *performance module,* comprising four blocks, which synthesizes the experiments and records the delay measurements. (I). *Database* block: The cloud DB can be used as a testing service if AUT data models are stored on the server DB. P_1, P_2, P_3, …, P_n denote one to *n* separate projects created in the DB. Each of these projects has an associated service account for authentication, and the credentials are stored in the JavaScript Object Notation (JSON) format. These service accounts can be used to call DB server APIs from our test environment. The credentials obtained via such a service account are used to authorize server requests. Load can then be applied through these projects by querying their entire DBs at different intervals. (II). *Client Simulator* block: This block consists of a *Java* application that handles the number of users or requests and queries the entire DB by building testing artifacts, including *databaseTranx.jar* files corresponding to the separate projects P_1, P_2, P_3, …, P_n in (I). These *jar* files are the Java executables that load the entire backend DB and are automatically adapted to changes in the AUT, usage patterns, and the available devices or OSs. (III). *Test Computer* block: This block consists of a bash script for test automation to simulate users, generate server workloads from the artifacts generated from blocks I and II, and test input parameters (loading parameters) corresponding to server loading, the details of which are discussed in the next subsection. It waits for input from block IV regarding the type of test (test device settings, Wi-Fi vs. LTE, etc.) before starting the simulation. (IV). *Test Mobile* block: This block runs the AUT (e.g., the case study HT Patient Helper app) based on the sequence of UI interactions defined in the test scenario. The simulation is initiated based on the parameters and depending on the load generated in block I. The input parameters for block III change based on the test scenario and the generated load or if the desired load corresponding to the simulation is not achieved. The AUT consists of delay analysis logic for recording T_{response}. Based on the *Test Mobile* OS, we use a corresponding SQLite library to store the recorded measurements from the simulations. We then perform further postprocessing on block III to analyze the results.

B. SIMULATION PARAMETERS AND ASSUMPTIONS

The developed methodology is characterized in terms of various derived parameters. The parameters passed as input to block III include the number of simultaneous applications, called *spawns,* denoted by *MAX_SPAWN*; the number of cycles for each spawn, called *iterations,* denoted by *MAX_ITERATIONS*; the number of threads per spawn (simultaneous *users*), denoted by *NUM_THREADS*; and the number of requests per thread (*requests*), denoted by *NUM_REQ_PER_THREAD*. The *load* parameter describes the amount of load observed at the server during the simulation based on the specified input parameters. These simulation parameters enable the user to easily tune the experiments for the required *Database* load. For a given required load and a selected test scenario, the choice of the values of *MAX_SPAWN*, *NUM_THREADS*, and *NUM_REQ_PER_THREAD* depends on the *Test Computer* machine power, the OS configuration, and the amount of heap memory allocated. Thus, there are no standard values for these parameters, and the maximum number of threads that can be generated from the *Test Computer* depends on the

Algorithm 1 Establishment of the Complete Simulation

Input: *databaseTranx.jar* files; initial values of *MAX_SPAWN, MAX_ITERATIONS, NUM_THREADS*, and *NUM_REQ_PER_THREAD*; step = 0
Output: Delay measurements in AUT's SQLite *DB*, T_{response}
repeat
Step 1: Backend DB loading:
 while step < *MAX_ITERATIONS* **do**
  Check for any open *Java* apps in the background
  if not then
   step ← step + 1
   spawn = 0
   while spawn < *MAX_SPAWN* **do**
    spawn ← spawn + 1
    nohup java –jar *databaseTranx.jar NUM_THREADS NUM_REQ_PER_THREAD*
   end while
  else
   sleep for 100 *ms*
  end
 end while
Step 2: Start simulation of AUT:
 Enter title of experiment based on test campaign and number of iterations
 AUT steps through the UI based on test scripts
 Store delay measurements in app.db
Step 3: Compute T_{response} from app.db using eq. 1
until end

computer's hardware and the OS. For the case study analysis, we found the desired combinations of these values during the experiments by monitoring the CPU, RAM, disk I/O, and network I/O of the *Test Computer*. Notably, although different combinations of the simulation parameters can theoretically result in the same amount of load, it affects the server in a different manner depending on the values of *NUM_THREADS* and *iterations*.

The following assumptions were adopted when running the performance simulations:

- The AUT is running in an isolated environment with no other apps in the background.
- Settings or modes from the same testing campaign (detailed in Section V.D) are not switched while simulations for an experiment with a particular setting are running—e.g., when experiments were running for the AUT with the *Test Mobile* device in the full battery and LTE modes, we did not switch to Wi-Fi mode in the middle of this process.

C. SIMULATION ALGORITHM

In this section, we show the complete simulation process flow of the methodology proposed in Sections III.A and III.B. The processes and blocks in Fig. 1 follow one another chronologically. As previously described, the process flow starts with the generation of testing artifacts as input, including loading specific files and setting parameters, and proceeds to obtain the final results, including the delays or T_{response} of the cloud backend service with respect to requests from the AUT. T_{response} is calculated in units of ms as follows:

$$T_{\text{response}} = T_{\text{received}} - T_{\text{sent}} \tag{1}$$

where T_{received} is the timestamp of the response received from the backend service and T_{sent} is the timestamp of the request sent from the AUT to the service. The time needed to display the result to the user via the UI of the AUT is not considered as part of the response time calculation. Importantly, the delay that occurs in app–cloud backend communication constitutes a major portion of the overall app response time, and this is a unique factor that separates the performance analysis of cloud-based mHealth apps from that of traditional apps. Algorithm 1 presents the pseudocode for running the full simulation based on the performance module and generated artifacts. Both *Steps* 1 and *3* are carried out on the *Test Computer*, and *Step* 2 is carried out on real test mobile devices. *Step 1* is an automation script that runs a backend-service-specific executable to generate load. As mentioned in the algorithm, the input parameters need to be adjusted in this step based on the required load, which depends on the *Test Computer* hardware specifications and OS. Hence, this step can be adapted to different backend services. The *Client Simulator* generates a *.jar* executable that is a packaged version of the project is specific to the project assets (backend DB structure), which we assemble to pass as input to *Step* 1. This file consists of the compilation output for the module in the project, the libraries included in the module dependencies, and a collection of resources such as images, video URLs, database nodes, individual files, directories, and archives. For server or cloud run and debug configurations that accept service account credentials and DB URLs, these artifacts can be generated and utilized for generating load. In *Step 2,* for the UI itself, we create automatic UI test scripts to mimic user interactions with the AUT through step-by-step button presses, taps, swipes and their order using Android- and iOS-specific test automation frameworks, as mentioned in Section V.

D. STATISTICAL ANALYSIS

For each experimental setting with varying load, device type, battery mode, and network combinations (shown in Table 4), the computed T_{response} values (equation (1)) are reported in the form of the mean, standard deviation, confidence interval, min–max statistics, median, and skewness for different load factors (increments of 25%). Based on the results of multiway analysis of variance (ANOVA) with Tukey's multiple comparison analysis, the statistical significance of the eligible independent variables is examined. A statistical significance threshold (α) of 0.05 was adopted and the analyses were performed using *R* [47].

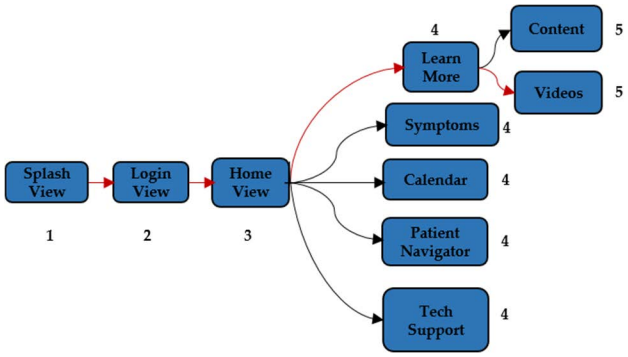

FIGURE 2. User interaction sequences in the app.

IV. CASE STUDY: PROTOTYPE APP DETAILS

The methodology proposed in this paper, and hence the evaluation process, requires the source code of the AUT. To this end, this section describes the case study app, with a brief justification of the design choices that led to the implementation and the structure of the DB. For the presentation of this case study, the adaptation of the testing methodology based on the proposed technique, and the reporting of the results, the current work has benefited from the case study research guidelines in [46].

A. OVERVIEW OF THE APP

The HT Patient Helper app leverages Firebase as the cloud DB server. User-specific information, symptoms, push notifications and videos reside in Firebase as corresponding data models. Firebase also simplifies storage and client–server communication. The educational content of the app is centrally deployed using GitHub pages for medical education management to provide users access to broader information about their symptoms and HT itself. Both the Android and iOS versions of the app use the same centralized Firebase DB nodes. Doctors, practical nurses (PNs) and administrative personnel are provided with a web-based interface to monitor the data received at the cloud DB. The reader is directed to [10] for additional details and features of and background information on the case study app. The sequence of user interactions (control flow) in both the Android and iOS versions of the app is shown in Fig. 2, where the arrows indicate the sequence of user taps in the test scenario. Fig. 2 shows a simple navigation sequence between different activities in the app, with a primary focus on the activities involved in a test scenario for the current study on performance analysis. Overall, the app supports 18 activities corresponding to different app functionalities, as described in [10]. Numbers 1 to 5 indicate the sequence. The Splash view appears momentarily at the launch of the app and simply displays the app's name and the organizations involved. In the Login view, users can log in to their accounts, following which they can see a list of buttons on the Home view corresponding to each feature of the app. The Learn More button offers further options for opening educational content and videos. The Symptoms button displays a list of breast cancer symptoms to track on a regular basis and presents options for obtaining a summary or more information about symptoms. The Calendar button opens the phone's calendar to make appointments and set reminders. The PN button provides information about the PNs and their contact information. The Tech Support button puts the user in contact with the technical support team. The test scenario consists of the sequence of button presses in the GUI indicated by red arrows.

The choice of the Android and iOS platforms is driven by the fact that they are the most dominant mobile OSs. This helps us to use a broad range of instrumentation tools associated with Android and iOS and understand the underlying systems. However, the ideas and techniques discussed in this paper can also be extended to other platforms. The apps utilize the same backend cloud-based DB Firebase (real-time), enabling developers to reduce their app development efforts and time due to the use of platform-specific services offered by each API. Firebase further enables data syncing across all client apps, making it a good choice for storing app-specific data models. Moreover, we selected Firebase for the case study analysis because it is easy to use and assists in deploying a framework suitable for prototyping similar mobile apps. For the testbed (Fig. 1), the choice of the components for blocks II (generation of testing artifacts) and III (automation script) makes the tests and analyses compatible across devices and generic for similar performance studies.

B. STRUCTURE OF THE FIREBASE PROJECT

The current study leverages Firebase's real-time DB core service as a DB backend-as-a-service. To enable the use of this service, the prototype Android and iOS mobile apps were created using the Firebase Android and iOS software development kits, respectively. The Firebase DB is a JSON NoSQL DB; thus, all data are stored in JSON format with a data structure consisting of a tree of hierarchical key–value pairs. To leverage Firebase optimizations for nonstructured data, the project DB was properly structured based on how the feature-specific data were to be saved and retrieved later. As an example, in Fig. 3, we show the structure of the *videos* node specific to the test scenario (Section V). Our goal was to maintain a balance between data normalization (shallow structure) and denormalization (deep structure) based on the usage of data corresponding to the app features (Section IV.A). When designing the data structure, we considered that querying information for one node in the JSON hierarchy automatically implies fetching data for all of its child nodes as well. It is hence more appropriate to keep the data structure as flat as possible. Thus, Fig. 3 presents a suitable way to store *videos* data, in which sections, videos, and their metadata are separated. The *sections* child node contains only metadata about each section or video category stored under the section's unique ID. The *videolist* child node consists of video metadata stored under the video's unique ID. These video metadata also include corresponding section

```
"videos" : {
   "sections" : {
     "HormoneTherapy" : {
       "mKey" : "HormoneTherapy",
       "mTitle_EN" : "Hormone Therapy",
       "mTitle_ES" : "Terapia Hormonal"
     }
     },
     "videolist" : {
     "english" : {

       "HormoneTherapy" : {
         "key0000" : {
           "mKey" : "key0000",
           "mSectionKey" : "HormoneTherapy",
           "mThumb" : "https://img.youtube.com/vi/hHQLXwLrLuo/0.jpg",
           "mTitle" : "Dr. Kate Lathrop",
           "mUrl" : "https://youtu.be/hHQLXwLrLuo"
         }
       }
       },
       "spanish" :{
       }
   }
}
```


FIGURE 3. Firebase project JSON tree flattened structure (showing videos node) that allows iterating through the list of videos.

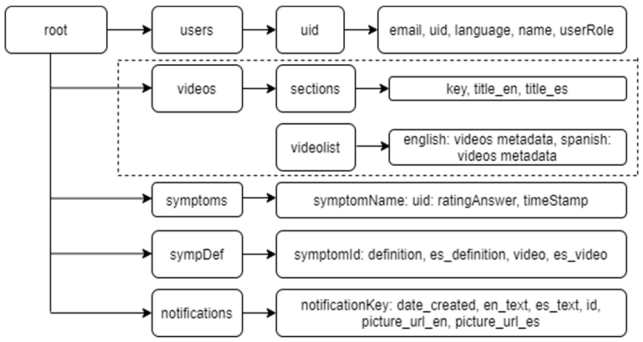

FIGURE 4. Firebase DB structure for the prototype apps.

IDs. Additionally, Fig. 4 shows the overall DB structure with other nodes corresponding to app features.

V. TESTING METHODOLOGY

This section presents the test scenario used to assess the performance of the Android and iOS prototype AUTs by approaching realistic and representative app usage patterns. We conducted three assessment experiments to observe the Firebase cloud performance and app capabilities in the test scenario: 1) *Firebase loading* (emulating many devices to add stress), 2) *device battery mode* effect on the network, and 3) *cellular (LTE) vs. Wi-Fi* effect on the performance of the apps' connections to the Firebase DB. For improved and seamless analysis of Firebase loading, we initially identified performance bottlenecks in the AUT that were not evident to the end user but affected overall performance. Brief overviews of the test platform, automated UI testing, test dataset, and test scenario are provided next.

A. TEST PLATFORM

The equipment used to conduct the experiments was as follows:

1. *Test Computer:* 2015 MacBook Pro 15'' (2.5 GHz i7 processor with 16 GB of 1600 MHz DDR3) running macOS Mojave.
2. *Test Mobile (Android):* Nexus 6P octa-core (4 × 1.55 GHz Cortex-A53 and 4 × 2.0 GHz Cortex-A57) running Android 8.0 Oreo.
3. *Test Mobile (iOS):* iPhone XS hexa-core (2 × 2.5 GHz Vortex and 4 × 1.6 GHz Tempest) running iOS 12.2.

B. UI AUTOMATION TESTING AND TEST SCENARIO

Scripting techniques for automated testing improve the accuracy per test case. This section describes the Android- and iOS-based mobile UI testing as part of the performance analysis. Manual testing is one of the easiest ways to perform UI testing. For the current work, however, manual testing would be problematic because it does not allow control over delays between views and is not consistent across tests. In automated UI testing, sets of scripts are defined to be rigorously executed in a quick, repeatable way regardless of the diverse functionalities of the AUT interface based on its features. Automated UI tests can reveal the presence of software issues in the transitions between consecutive releases of an app. Although the scope of our work does not include solely functional UI testing as part of the performance study process, automated tools for input generation and UI recording and other open-source multiplatform automated testing tools (e.g., Calabash and Appium[5]) often rely on underlying UI automation frameworks (e.g., UI Automator, Espresso, and XCTest (iOS)) [41]–[44]. Hence, we used the state-of-the-art vendor-provided UI Automator[6] testing framework along with Espresso[7] and XCTest[8] for testing and generating repeatable test scripts for Android and iOS, respectively. These frameworks are specifically used to simulate user interactions with an AUT and run UI tests in an automated and repeatable way. Each test script includes a complete path from an initial to a final state in the user interaction sequence. We used the APIs offered by the abovementioned frameworks to build the GUI tests and implemented test scripts emulating user interactions such as button clicks and scroll. We manually entered the series of actions (as shown in Fig. 5) to be performed on different GUI components in the test scripts.

C. TEST DATASET

For the test data, we used a massive video dataset for Firebase loading. We adopted publicly available videos from YouTube.com and other video sites and used the command line tool youtube-dl[9] to download the list of video URLs. As previously noted, our video data reside in a Firebase DB. We designed the data model to include videos based on different sections. The *videos* node in Firebase has nested *sections* and *videos* nodes. The *sections* node has section names as keys, each of which is associated with videos in the *videos* node. Videos can be identified by section names, and

[5]http://appium.io/
[6]https://developer.android.com/training/testing/ui-automator
[7]https://developer.android.com/training/testing/espresso/
[8]https://developer.apple.com/documentation/xctest
[9]https://github.com/ytdl-org/youtube-dl

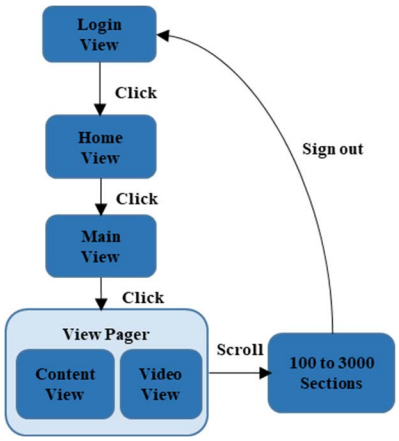

FIGURE 5. Test scenario for the retrieval of educational sections and videos.

the entry for each video includes the thumbnail image, URL, and video title as keys. Additional details about exporting the dataset are presented in [25].

D. EXPERIMENTAL DESIGN

This section describes the three experiments conducted in the aforementioned test scenario. The scenario consisted of users signing into the app, accessing the AUT's video view, and retrieving sections.

1) FIREBASE LOADING

Since the sections are retrieved from the Firebase DB, we hypothesized that as the Firebase load increases, the response time will also increase. Therefore, we conducted three sets of experiments to observe the response time variations; these experiments are described next, followed by Android and iOS comparisons. For the current app, we used the free version of Firebase, which allows only 100 simultaneous users, 10 GB of bandwidth, and 10 GB of storage. We started to load Firebase by querying the entire *videos* DB at different time intervals. We created a separate Java application as described in Section III.A to control different threads to simulate a user submitting a request to Firebase. A bash test automation script was then used to control the numbers of spawns, iterations, simultaneous users, and requests, as described in the simulation flow [Algorithm 1]. The Firebase console load graph shows only the DB usage when processing app requests over a 1-minute interval, representing the highest load in a given hour. As such, if the target load is 25% and the test run displays a load of 30%, there is a wait time of an hour before another test can be executed. To accelerate this process, we created three separate Firebase projects—P_1, P_2, and P_3—for the app's performance module.

To enable proper control over the delays between views and consistency between tests, the performance analysis of the AUT's section retrieval control flow from no load to full load on the Firebase DB (0–100%) in increments of 25% also models the delay on LTE and Wi-Fi connections while considering energy characteristics (device battery mode). Fig. 5 shows the sequence of user interactions performed by the test scripts in detail. The simulations followed this path in every iteration.

TABLE 2. Network specifications.

LTE	Wi-Fi
12-30 Mb/s down	200 Mb/s down
1-5 Mb/s up	10 Mb/s up
AT&T	Spectrum

2) DEVICE BATTERY TEST

An increase in service response time with section retrieval through the app directly affects the user experience. The response time is affected by device resource utilization, i.e., the energy consumption of different system components. The device battery also contributes to the response time. We therefore performed a test by switching between the *regular* and *power-saving (PS)* modes to analyze the effect of the device battery mode in the presence of varying loads.

3) CELLULAR LTE VS. WI-FI TEST

This test assessed the impact of the available mobile wireless network connectivity on the app's interactions and response time for the test scenario, again in the presence of dynamic Firebase loads. We conducted a study of these interactions and their typical impact in terms of the 95^{th}-percentile response time for Android and iOS using a combination of measurements in the regular and PS battery modes. The uplink and downlink data rate specifications of the LTE and Wi-Fi networks for each service provider are shown in Table 2.

4) SIMULATING USERS

The Firebase loading test required simulating users and their interactions based on the test scenario discussed in Section V.B. Fig. 6 shows the complete listener lifecycle (sections and videos). It utilizes a singleton pattern, a design pattern that restricts the instantiation of a class to one object, to maintain a list of video and section listeners. Each listener holds a strong reference to the host *fragment* in Android or a *view controller* in iOS and must be released from memory to avoid memory leaks. The scenario is further specified as follows:

1. Sections are paginated by 10 and are loaded when the user first enters this view or is at the bottom of the view. Each page contains a single listener object that is stored in a singleton object.
2. When a section is clicked, the section key is used to reference the associated videos in the node structure from Firebase creating a new listener. Again, the listener is stored in a singleton object.

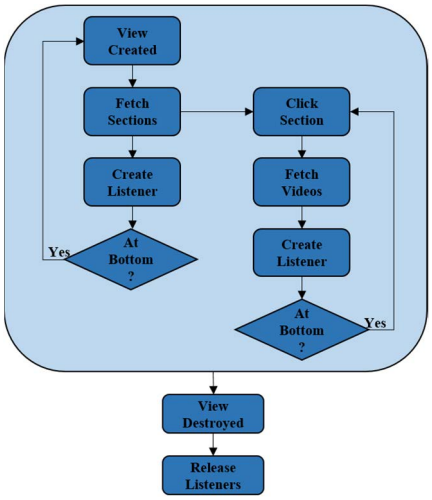

FIGURE 6. Listener lifecycle.

3. Sections and videos need to be indexed by location and to avoid duplicate listeners for the video list.
4. When a view is destroyed, the singleton object releases all associated section and video listener references to avoid strong references to the associated objects.

We found that using automated UI testing frameworks to simulate user interactions and conduct multiple repeatable tests was slow and caused the app to crash after a few logins due to memory leaks. Hence, we used the Android Profiler[10] with the LeakCanary library[11] and iOS Instruments[12] to track such leaks and found mostly Firebase-based leaks. These leaks were due to strong references to an object that were not eventually released in cases where a Firebase listener was created. Such strong references [26] to an activity or view controller must be removed through the garbage collection (GC) and automatic reference counting (ARC) memory management mechanisms in Android and iOS, respectively, to provide a consistent load on Firebase and simulate user actions. If an object is no longer in use, such mechanisms recover the corresponding memory and allow it to be reused for future object allocation. Such memory management must be handled at compile time rather than at runtime to avoid memory and performance overheads. Each experiment involved fetching the section list from Firebase in the presence of an applied load. The time taken to retrieve the section was then recorded locally in an SQLite DB and displayed after completion. These processes were the same for both Android and iOS experiments conducted in different battery and mobile wireless network modes.

[10]https://developer.android.com/studio/profile/android-profiler

[11]https://github.com/square/leakcanary

[12]https://help.apple.com/instruments/mac/current/#dev7b09c84f5

TABLE 3. List of testing tools and APIs.

Category	Tools
CPU utilization, Memory usage	Android profiler, Instruments (iOS)
Memory leaks	Leak detector (Leak Canary)
Testing frameworks	UIAutomator, Espresso (Android), XCTest (iOS)
Image loading and caching	Glide (Android), Nuke (iOS)
Delay modeling (*SQLite* DB)	Persistence libraries: Room (Android), SQLite.swift (iOS)
$T_{response}$	Bash, Java and Python scripts

Information about the number of simultaneous connections or users' retrieval of videos is available by querying the complete DB node, and a more accurate overview of the DB's overall performance is available through the *Usage* tab on the Firebase console. However, this means of gathering information is not suitable for troubleshooting potential performance issues or understanding fine details. Our proposed performance module provides the functionality of observing performance metrics and the impact of different AUT parameters and device settings over time. The real-time DB integration with the performance module offers the deepest level of granularity. The performance measurements were completed over the course of 2 minutes for each test and are discussed next.

VI. EXPERIMENTAL RESULTS

This section describes the measurement results from the experiments conducted on real Android and iOS devices to evaluate our proposed performance methodology. Table 3 [26] shows the categories of metrics, libraries, and testing frameworks we used to monitor the app interactions, save data to the SQLite DB, model delays, implement video listeners for the test scenario, and visualize the effect of the Firebase load on $T_{response}$ for the Android and iOS versions of the AUT. Both UI Automator and Espresso are part of the official Android instrumentation framework. UI Automator gives Espresso the ability to test the GUI of an AUT along with the device status and performance. The combination of these tools also provides the flexibility to test multiple apps at the same time and perform operations on the AUT GUI and the device, e.g., turning Wi-Fi or LTE on or off or changing the battery mode to PS. To validate the performance methodology, clarify the extent to which the experimental performance data are indicative of representative behaviors and analyze the impact of the operating conditions, we conducted a 95% confidence interval and multiway ANOVA test. The reliability metrics, such as confidence interval values, and other statistics (Tables 5 and 6) indicated similar behavior of the response time with respect to an increase in server load under the considered operating conditions on both Android and iOS devices. Thus, we show the trace plots for the experiments in which we observed slight differences in the pattern.

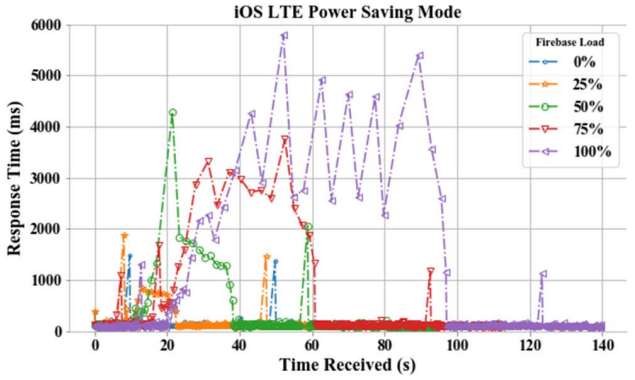

FIGURE 7. iOS response time plot with LTE in PS mode.

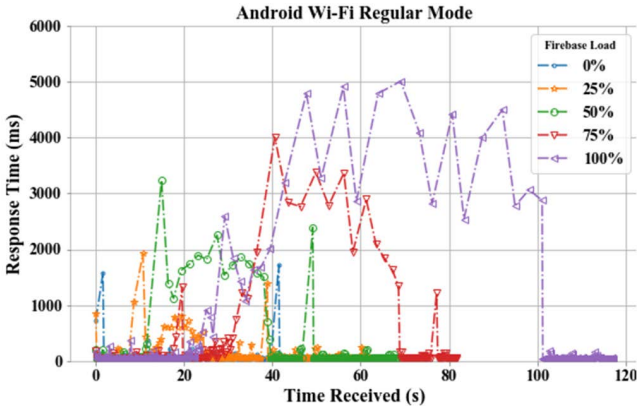

FIGURE 8. Android response time plot with Wi-Fi in regular mode.

A. LTE CONNECTION WITH REGULAR BATTERY MODE

In this subsection, we report simulations of the AUT run in the test scenario with an LTE connection under the regular device battery mode. The response times for Android and iOS, respectively, for retrieving videos monotonically increase with increasing load on the Firebase DB (no load, 25%, 50%, 75%, and full load), causing an immediate effect on T_{response} and thus influencing the UI responsiveness experienced by the user. For most of the simulation duration under each load, Android and iOS performed similarly.

B. LTE CONNECTION WITH POWER SAVING MODE

In this subsection, we report experiments performed with the Android and iOS versions of the AUT with an LTE connection and the device battery in the PS mode. Overall, we observed similar behavior as in Section VI.A, i.e., an increasing response time with an increase in the Firebase load, with some anomalies under loads of 50 and 75% for a short period of time [Fig. 7]. The reason could be inconsistent network behavior leading to a rapid increase in the response time under any load. These spikes were due to the load, and the total duration of the spikes under each load occurred at almost the same time during the simulation for each OS. For the full-load case, the maximum peak response time and spike duration occurred for 20 seconds for both the Android and iOS AUTs.

C. WI-FI CONNECTION WITH REGULAR AND POWER SAVING MODES

This subsection summarizes the experiments performed on the AUT with a Wi-Fi connection and with the device battery in the regular and PS modes on both Android and iOS. Here, we again observed that the response time increased with the Firebase load. However, due to noise and network inconsistencies, the 50% load scenario showed a greater T_{response} than the 75% load scenario for a short initial duration (as shown in Fig. 8 for the Android/Wi-Fi/regular mode of operation).

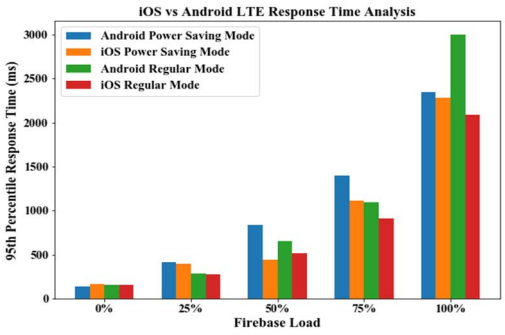

FIGURE 9. Android vs iOS LTE 95th percentile plot with varying load.

D. ANDROID VS. IOS: 95TH PERCENTILE T_{RESPONSE}

Finally, we show the 95$^{\text{th}}$-percentile response times for the above-described simulation conditions on Android and iOS under Firebase loading. We performed these tests to identify the T_{response} that most users would encounter, excluding outliers, to obtain a clearer picture of the characteristics of the AUT and augment the aforementioned analysis. In each scenario shown in Fig. 7 and Fig. 8, the response time was directly affected by the load regardless of the network connectivity mode or device battery mode. Moreover, the time needed to clear each load increased due to the increase in the response time. It can be intuitively expected that the battery mode should not impact the response time, as the app is running in the foreground and the tasks of sending and receiving are not CPU intensive. Both the iOS and Android versions of the AUT exhibited similar response times, summarized in terms of the 95$^{\text{th}}$ percentile of T_{response} in Fig. 9 and Fig. 10. It is also clear that there is no single winner here.

While T_{response} is a key performance parameter, as expected and as observed from the effect of Firebase loading during UI simulations for the test scenario, our findings also highlight the importance of the complexity of the DB queries and the data model used in the cloud DB.

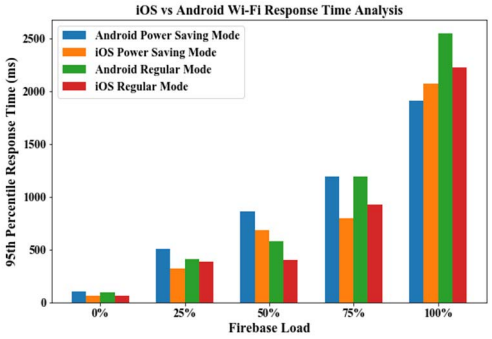

FIGURE 10. Android vs iOS Wi-Fi 95th percentile plot with varying load.

TABLE 4. Experiments notations.

Experiment	Notation
Android with LTE in PS mode	A/LTE/PS
Android with LTE in *Regular* mode	A/LTE/R
Android with Wi-Fi in PS mode	A/WiFi/PS
Android with Wi-Fi in *Regular* mode	A/WiFi/R
iOS with LTE in PS mode	iOS/LTE/PS
iOS with LTE in *Regular* mode	iOS/LTE/R
iOS with Wi-Fi in PS mode	iOS/WiFi/PS
iOS with Wi-Fi in *Regular* mode	iOS/WiFi/R

E. CONFIDENCE INTERVALS AND RESULTS ANALYSIS

This section presents T_{response} confidence intervals and statistical analysis to characterize the tail behaviors, the impact of different operating conditions, and the uncertainties underlying the measured response times. Tables 5 and 6 present descriptive statistics that describe the quality characteristics of the experimental data collected from the simulation experiments. Specifically, Table 5 shows the mean, standard deviation, and 95% confidence intervals. From Table 6, we observe that the mean is always greater than the median, indicating highly positively skewed data with many response times lying in the tail to the right of the mean. It should be noted that the t-interval and Z-interval calculations for the mean result in similar values. Therefore, given the nonnormal distribution of the data and a sufficiently large sample size ($sz > 30$), the confidence interval of T_{response} for each experiment in Table 4 can be derived as follows:

$$intv\left(T_{\text{response}}\right) = \left(T_{\text{response\_mean}} \pm t_{\frac{\alpha}{2},sz-1}\left(\frac{sdv}{\sqrt{sz}}\right)\right) \quad (2)$$

where $T_{\text{response\_mean}}$ denotes the mean of the experimental response times, sdv is their standard deviation, sz is the sample size, t is the t-interval, and α is the confidence level.

In the following, we further investigate the impact of different independent variables (device type, battery mode, and network setting) on the response time (and the overall performance of the case study app) through a multiway ANOVA method. We utilize ANOVA to determine how much of the variability in T_{response} is explained by each of the different variables (load factor, device type, network setting and battery mode). From the p values obtained for each of these independent variables, we have sufficient statistical evidence ($p < 0.0001$) that T_{response} is different for different load factors, device types, and network settings. Since ANOVA highlights the variations in the mean value of a distribution, finding statistical significance, i.e., $p < 0.05$, implies 95% confidence that the different values are indicative of representative differences. In contrast, there is insufficient statistical evidence ($p = 0.9826$) to identify a significant impact of the battery mode. Moreover, with multiple independent variables, two or more variables may have interacting effects on T_{response}. Therefore, we performed Tukey's pairwise comparison test and obtained very low p values (<0.001) for the interaction effects between the load factor and the device type and between the device type and the network setting. We did not find statistical evidence of an interaction effect between the device type and the battery mode ($p < 0.01$).

F. THREATS TO VALIDITY

The validity of the proposed approach, testing methodology and case study presented in this paper is not without threats, as in any experimental and empirical study. There are several factors that may affect the validity of the presented methodology and results.

1) EXTERNAL VALIDITY

A potential threat is that our study included only a medical-reference-based mHealth app, which may be tested and designed differently than other mHealth apps that exchange biomedical information. Therefore, our conclusions may not apply to all mHealth apps. However, it is important to note that our results are still representative of most mHealth apps for medical adherence and disease management as well as other mobile apps with multimedia-intensive features that leverage cloud backend services. Furthermore, to generate testing artifacts specific to a particular backend service, elements of the cloud or server configuration, such as the project service account credentials, DB structure, and location URLs, should be considered for utilization in the proposed approach.

Most of our analysis focused on a Firebase cloud DB, which could have different usage patterns than other types of on-premises and cloud DBs. Consequently, our results may not generalize beyond similar NoSQL cloud DBs (e.g., Couchbase Lite DB and Amazon Web Services Dynamo DB) due to the differences in the DB API designs. Additionally, the results of our study can only be generalized to Android and iOS mobile platforms. Nevertheless, step 1 of Algorithm 1 can be adapted to mobile apps utilizing Firebase as well as apps that leverage architecturally similar mobile backends-as-a-service, from which we could generate testing artifacts analogous to those discussed in the current study. Hence,

TABLE 5. Experimental performance results, and confidence intervals.

			PS				Regular			
Network	Device	Load factor	Sample size	Mean (ms)	SD (ms)	CI (95%) (ms)	Sample size	Mean (ms)	SD (ms)	CI (95%) (ms)
LTE	Android	0	334	112.95	125.66	[99.40 - 126.50]	334	121.45	147.05	[105.60 - 137.30]
		25	334	145.75	226.89	[121.30 - 170.21]	334	144.96	191.48	[124.32 - 165.61]
		50	289	192.28	413.73	[144.30 - 240.27]	319	170.32	281.79	[138.19 - 202.44]
		75	320	226.03	558.89	[164.47 - 287.59]	299	234.37	604.76	[166.56 - 302.17]
		100	287	353.62	957.97	[242.12- 465.12]	327	358.96	1049.07	[240.95 - 476.97]
	iOS	0	334	126.03	105.32	[114.68 - 137.38]	334	122.57	85.62	[113.34 - 131.79]
		25	334	145.8	170.32	[127.44 - 164.16]	334	144.43	170.19	[126.08 - 162.77]
		50	334	187.37	352.99	[149.32 - 225.42]	334	181.67	254.25	[154.27 - 209.08]
		75	334	241.28	523.67	[184.83 - 297.73]	334	259.7	569.04	[198.36 - 321.04]
		100	334	328.42	831.24	[238.81 - 418.02]	334	348.07	1037.3	[236.25 - 459.89]
Wi-Fi	Android	0	334	67.22	101.76	[56.25 - 78.19]	334	66.51	130.85	[52.40 - 80.61]
		25	334	95.98	175.55	[77.06 - 114.91]	334	101.39	181.92	[81.78 - 120.99]
		50	319	147.56	354.02	[108.49 - 186.62]	309	143.75	398.01	[99.13 - 188.38]
		75	299	208.95	608.46	[139.58 - 278.31]	307	186.49	536.49	[126.15 - 246.84]
		100	327	284.49	872.2	[189.47 - 379.53]	317	286.84	863.77	[191.23 - 382.44]
	iOS	0	334	51.79	78.05	[43.38 - 60.21]	334	55.96	116.88	[43.36 - 68.56]
		25	334	83.98	169.58	[65.69 - 102.26]	334	87.44	155.05	[70.73 - 104.15]
		50	334	136.99	350.65	[99.18 - 174.78]	334	133.79	384.09	[92.39- 175.20]
		75	334	180.46	580.81	[117.85 - 243.07]	334	172.55	507.39	[117.85 - 227.24]
		100	334	273.5	865.91	[180.16 - 366.84]	334	267.29	833.37	[177.46 - 357.13]

as per the best practices described in [45], we have addressed generalization considerations to balance generality with the development of a practical case study app built using Firebase as the cloud backend DB. Moreover, all software engineering studies suffer from the variability of real-world apps, and thus, the issue of generalizability cannot be completely resolved.

2) INTERNAL VALIDITY

Android and iOS are both continuously running parallel tasks that may affect data access delays and the responsiveness of the UI. For that reason, we ran our experiments in a controlled setting, as mentioned in Section III.B. UI interactions normally stimulate internal tasks in the mobile app running in the foreground. While conducting testing using automated test scripts, the mobile app was improved by addressing memory leaks in the implementation to prevent any side effects of UI events. Additionally, the time needed for the UI to display the requested data was not considered in any experiment. Thus, these experiments measured only the data access delays or service response times. We utilized vendor-specific test automation frameworks to manually write test scripts specific to the sequence of user interactions to be repeatedly performed in each experiment. Hence, the AUT block will require continuous effort in updating the test scripts to match any new design of the app.

The performance aspects considered, and the metrics calculated in this approach may also not be exhaustive or representative of everything that characterizes performance-based nonfunctional requirements that need to be tested. Nevertheless, these aspects and metrics are essential for testing the primary aspects of a mobile apps' performance.

VII. DISCUSSION

Mobile network performance is a crucial element in mobile–cloud communications and in dealing with delay-sensitive mobile apps. In recent performance studies regarding LTE networks [28] and crowdsourced measurements [29] conducted on mobile devices using measurement apps, the network throughputs and latencies in one or both of the download and upload directions have been measured as metrics. User- and network-level performance can be reliably inferred from these metrics to properly characterize the user experience. However, they are based on standard

TABLE 6. Tail behavior statistics.

			PS				Regular			
Network	Device	Load factor	Median (50^{th} %ile) (ms)	Min (ms)	Max (ms)	Skewness	Median (50^{th} %ile) (ms)	Min (ms)	Max (ms)	Skewness
LTE	Android	0	98	69	1514	9.81	101	67	2100	10.27
		25	96	72	2168	5.99	101	76	1832	5.58
		50	89	65	4373	5.89	94	63	1803	4.18
		75	83	59	3503	4.22	86	64	4050	4.66
		100	95	66	5951	4.09	93	67	7244	4.18
	iOS	0	112	84	1480	11.5	111	75	1244	10.62
		25	107	76	1886	6.07	107	79	1994	6.64
		50	110	79	4285	6.87	115	78	1605	4.28
		75	112	83	3759	4.58	121	80	4037	4.79
		100	107	78	5794	4.31	104	67	9922	5.17
Wi-Fi	Android	0	51	42	1303	9.64	49	40	1718	11.03
		25	51	40	1933	5.72	52	40	1923	5.67
		50	48	39	3372	4.86	45	38	3238	4.7
		75	51	41	4564	4.49	46	37	4004	4.73
		100	47	39	5808	4.38	45	34	5010	3.98
	iOS	0	45	34	1368	15.21	45	36	1697	12.89
		25	43.5	34	1712	5.63	48	35	1339	4.79
		50	47	35	3365	4.99	46	34	3288	4.95
		75	44	34	4495	4.86	46	34	3760	4.8
		100	47	34	5822	4.42	46.5	34	4890	4.09

measurement apps such as Ookla Speedtest[13] and MobiPerf and the network datasets collected by those apps. In contrast, our own measurement-based performance study employed similar network-related measurements but relied on the observed communication between our developed case study mHealth app and the corresponding Firebase cloud DB. For the proposed performance methodology, our analysis began with an examination of the general characteristics of mobile–cloud performance in terms of the response time for video retrieval based on LTE vs. Wi-Fi network connections with the user device operating in the regular and PS battery modes, as discussed in Section VI.

These characteristics were further revealed through the delay traces observed over 2-minute simulation durations for each test condition under varying Firebase loads. In practice, the service response time has a great impact on the user experience and, hence, on the satisfaction with the underlying service. Overall, we did not observe significant differences between Android and iOS devices operating in different battery modes. However, we did detect a difference when changing between the LTE and Wi-Fi mobile networks. To further examine the measured response times and for validation, we turn to a broader view of the response times for Android and iOS devices under different Firebase loads, as shown in Tables 5 and 6. These tables show the differences in download performance between the LTE and Wi-Fi cases. Before analyzing these data further, we present abbreviated notations for the conducted experiments in Table 4. The values in Tables 5 and 6 (means, standard deviations, confidence intervals, and skewness-related statistics) are representative of the average download performance of AUT–cloud communication on Android and iOS devices, respectively. The response time clearly increased with the Firebase load in all conducted experiments, while the battery mode had no significant effect for either LTE or Wi-Fi (see Section VI.E for the results of Tukey's pairwise test).

Comparisons, however, reveal that the A/Wi-Fi/R and A/Wi-Fi/PS configurations (in general, an Android device on Wi-Fi) provided better average performance for nearly every Firebase load than the A/LTE/R and A/LTE/PS configurations (in general, an Android device on LTE) did. In particular, the average zero-load response times for A/Wi-Fi/R and A/Wi-Fi/PS were 66.51 ms and 67.22 ms, respectively, whereas those for A/LTE/R and A/LTE/PS were 121.45 ms and 112.95 ms, respectively. This substantial difference in download performance of 50–70 ms between Wi-Fi and LTE persists across different load factors, and we can also observe a similar trend in the iOS case.

[13]https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest&hl=en_US

As seen from Table 5, for most of the experiments, the 95% confidence intervals for the 50% and 75% load factors overlap, indicating a nonsignificant effect of the load factor on the response time. Additionally, the variance for the 75% load factor seems high. With regard to the distribution of the experimental response time data collected from the simulations, we also calculated lower confidence intervals (90%, 85%, and others, not included in the manuscript). The data points in the long tail of the distribution were ignored when calculating lower confidence intervals. Considering that the skewness is > 1 (Table 6 indicates a highly skewed distribution), the response time cannot be assumed to follow a normal distribution. Interestingly, as observed from the simulation traces in Section VI and the statistics in Section VI.E, while the response time did not appear to be significantly affected by the mobile OS, Android devices recorded higher response time measurements than iOS devices in nearly every scenario under Firebase loading, as shown in Tables 5 and 6. Given that we were measuring AUT–cloud communication and leveraging the same Firebase API for each OS, it is possible that the higher T_{response} measurements for Android devices may be due to some buffering of the network data or differences in design between the APIs. Moreover, the Firebase DB receives many initial requests. The reason for the presence of spikes in Fig. 7 and Fig. 8 is the increasing load and the increasing request rates while earlier video retrieval requests are being served by the DB. Additionally, neither Firebase nor the case study app implementation has a default scaling mechanism. We also learned that the response time values are dependent on the cloud service provider and the resource availability of the service plan chosen for the project. As mentioned in Section V.D.1, these experiments were conducted on a free plan. If we were to scale up, then the simulations would need to be rerun to obtain a relevant set of independent variables for generating loads of 25%, 50%, etc., tailored to the service provider. Hence, it is also worth noting that T_{response} can be affected by many performance variables in the cloud environment, such as the network type, the type of cloud-hosted DB, and variations in hardware performance.

VIII. CONCLUSION AND FUTURE WORK

This paper has presented a technique for testing- and measurement-based performance studies for Android and iOS apps that leverage third-party cloud backend services. In this study, we performed a series of experiments using Android and iOS versions of the prototype mHealth HT Patient Helper app under various load conditions of the cloud-based backend DB (Firebase) with different modes of device battery usage and mobile network connection. The actual trace plots of the response times with respect to the simulation time and the reliability metrics obtained from a statistical analysis showed that the performance was consistent between Android and iOS in all experiments and that T_{response} increased with an increasing Firebase load. We found that the response time performance was not directly associated with the mode of battery operation (PS or regular). However, our study revealed certain characteristics of Wi-Fi and LTE mobile networks in mobile–cloud communication. The average Wi-Fi response time measurements were lower than the corresponding LTE measurements by at least a factor of two. This indicates that when retrieving large chunks of data, if the signal strength is good, Wi-Fi is more efficient. We also found that the mean T_{response} measurements for Android devices were slightly higher than those for iOS devices due to buffering, differences in the design of the Firebase APIs, or both. The confidence intervals clarified to what extent the mean values are indicative of representative differences and the potential impact on tail behaviors. This detailed analysis could provide some guidance when realizing quality-sensitive mobile applications with a cloud backend.

The testing setup and the case study app do not claim commonality but provide a sense of possible impacts caused by various factors for a representative mobile apps with multimedia data processing utilizing cloud services. At the same time, the introduced technique and testing methodology with mobile device settings is applicable for other applications. In future research, to accurately measure performance, we plan to explore a wide range of key performance indicators that are relevant to nonfunctional requirements, such as service-oriented metrics (availability) and efficiency-oriented indicators (throughput and capacity), along with other cloud platforms in diverse geographical locations.

REFERENCES

[1] (2016). *Smartphone Users Worldwide 2016–2027*. Accessed: Mar. 20, 2022. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] J. Gao, X. Bai, W. T. Tsai, and T. Uehara, "Mobile application testing: A tutorial," *IEEE Comput.*, vol. 47, no. 2, pp. 26–35, Jan. 2014.

[3] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 399–410.

[4] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *J. Syst. Softw.*, vol. 117, pp. 334–356, Jul. 2016.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[6] M. Linares-Vasquez, M. White, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "Mining Android app usages for generating actionable GUI-based execution scenarios," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 111–122.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2012, pp. 258–261.

[8] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proc. 10th Work. Conf. Reverse Eng. (WCRE)*, 2003, pp. 260–269.

[9] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, "AMOGA: A static-dynamic model generation strategy for mobile apps testing," *IEEE Access*, vol. 7, pp. 17158–17173, 2019, doi: 10.1109/ACCESS.2019.2895504.

[10] P. Chalela, E. Munoz, D. Inupakutika, S. Kaghyan, D. Akopian, V. Kaklamani, K. Lathrop, and A. Ramirez, "Improving adherence to endocrine hormonal therapy among breast cancer patients: Study protocol for a randomized controlled trial," *Contemp. Clin. Trials Commun.*, vol. 12, pp. 109–115, Dec. 2018.

[11] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generat. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, 2009.

[12] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing database as a service," in *Proc. 18th Int. Conf. Data Eng.*, Feb. 2002, pp. 29–38.

[13] L. Zhao, S. Sakr, and A. Liu, "A framework for consumer-centric SLA management of cloud-hosted databases," *IEEE Trans. Services Comput.*, vol. 8, no. 4, pp. 534–549, Jul./Aug. 2015.

[14] C. Gyorödi, R. Gyorödi, and R. Sotoc, "A comparative study of relational and non-relational database models in a web-based application," *Int. J. Adv. Comput. Sci. Appl.*, vol. 6, no. 11, pp. 78–83, 2015, doi: 10.14569/IJACSA.2015.061111.

[15] M. Abourezq and A. Idrissi, "Database-as-a-service for big data: An overview," *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, no. 1, pp. 157–177, 2016, doi: 10.14569/IJACSA.2016.070124.

[16] W. Al Shehri, "Cloud database database as a service," *Int. J. Database Manage. Syst.*, vol. 5, no. 2, pp. 1–12, Apr. 2013, doi: 10.5121/ijdms.2013.5201.

[17] S. D. Bijwe and P. L. Ramteke, "Database in cloud computing-database—A-service (DBaas) with its challenges," *Int. J. Comput. Sci. Mobile Comput.*, vol. 4, no. 2, pp. 73–79, 2015.

[18] R. Gyorodi, M. I. Pavel, C. Gyorodi, and D. Zmaranda, "Performance of OnPrem versus azure SQL server: A case study," *IEEE Access*, vol. 7, pp. 15894–15902, 2019.

[19] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in Android applications," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 389–398.

[20] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 1013–1024.

[21] A. Nistor and L. Ravindranath, "SunCat: Helping developers understand and predict performance problems in smartphone applications," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2014, pp. 282–292.

[22] B. M. C. Silva, J. J. P. C. Rodrigues, I. de la Torre Díez, M. López-Coronado, and K. Saleem, "Mobile-health: A review of current state in 2015," *J. Biomed. Inform.*, vol. 56, pp. 265–272, Aug. 2015.

[23] P. D. Kaur and I. Chana, "Cloud based intelligent system for delivering health care as a service," *Comput. Methods Programs Biomed.*, vol. 113, pp. 346–359, Jan. 2014, doi: 10.1016/j.cmpb.2013.09.013.

[24] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Softw.*, vol. 32, no. 3, pp. 70–77, May 2015, doi: 10.1109/MS.2014.50.

[25] G. Rodriguez, D. Inupakutika, S. Kaghyan, D. Akopian, P. Lama, P. Chalela, and A. G. Ramirez, "Performance assessment of mHealth apps," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2020, pp. 1–7, doi: 10.1109/PerComWorkshops48775.2020.9156198.

[26] M. Jun, L. Sheng, Y. Shengtao, T. Xianping, and L. Jian, "LeakDAF: An automated tool for detecting leaked activities and fragments of Android applications," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 2017, pp. 23–32.

[27] J. Qian and X. Zhou, "Inferring weak references for fixing Java memory leaks," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, doi: 10.1109/ICSM.2012.6405323.

[28] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, 2012, pp. 225–238.

[29] W. Li, D. Wu, R. K. C. Chang, and R. K. P. Mok, "Toward accurate network delay measurement on Android phones," *IEEE Trans. Mobile Comput.*, vol. 17, no. 3, pp. 717–732, Mar. 2018.

[30] I. C. Morgado and A. C. R. Paiva, "The iMPAcT tool: Testing UI patterns on mobile applications," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 876–881.

[31] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon, "A pattern-based approach for GUI modeling and testing," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2013, pp. 288–297.

[32] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl, "Anatomizing application performance differences on smartphones," in *Proc. 8th Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, San Fransisco, CA, USA, 2010, pp. 165–178.

[33] H. Kim, B. Choi, and W. E. Wong, "Performance testing of mobile applications at the unit test level," in *Proc. 3rd IEEE Int. Conf. Secure Softw. Integr. Rel. Improvement*, Jul. 2009, pp. 171–180.

[34] V. L. L. Dantas, F. G. Marinho, A. L. da Costa, and R. M. C. Andrade, "Testing requirements for mobile applications," in *Proc. 24th Int. Symp. Comput. Inf. Sci. (ISCIS)*, Sep. 2009, pp. 555–560.

[35] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Proc. 7th Int. Workshop Autom. Softw. Test (AST)*, 2012, pp. 29–35.

[36] Y. Kang, Y. Zhou, M. Gao, Y. Sun, and M. R. Lyu, "Experience report: Detecting poor-responsive UI in Android applications," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng.*, Oct. 2016, pp. 490–501.

[37] P. Zhang and S. G. Elbaum, "Amplifying tests to validate exception handling code: An extended study in the mobile application domain," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 1–28.

[38] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in Android applications," in *Proc. 1st Int. Workshop Eng. Mobile-Enabled Syst. (MOBS)*, May 2013, pp. 1–6.

[39] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyande, and J. Klein, "Automated testing of Android apps: A systematic literature review," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 45–66, Mar. 2019.

[40] M. K. Kulkarni and A. Soumya, "Deployment of calabash automation framework to analyze the performance of an Android application," *J. Res.*, vol. 2, no. 3, pp. 70–75, 2016.

[41] M. Linares-Vasquez, "Enabling testing of Android apps," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 763–765.

[42] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2014, pp. 204–217.

[43] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented evolutionary testing of Android apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 599–609.

[44] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu, "Capture-replay testing for Android applications," in *Proc. Int. Symp. Comput., Consum. Control*, Jun. 2014, pp. 1129–1132.

[45] R. Wieringa and M. Daneva, "Six strategies for generalizing software engineering theories," *Sci. Comput. Program.*, vol. 101, pp. 136–152, Apr. 2015.

[46] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.

[47] M. J. Crawley, *Statistics: An Introduction Using R*. Hoboken, NJ, USA: Wiley, 2014.

DEVASENA INUPAKUTIKA (Graduate Student Member, IEEE) received the B.Tech. degree from MITS University, India, and the M.Sc. degree from the University of Salford, Manchester, U.K. She is currently pursuing the Ph.D. degree in electrical engineering with The University of Texas at San Antonio (UTSA). Her current research interests include wearable technology and the IoT and web and mobile application development.

GERSON RODRIGUEZ received the B.S.E.E. degree from The University of Texas at El Paso and the M.S. degree in computer science from The University of Texas at San Antonio (UTSA). He was a Project Engineer at DuPont. His research interests include mobile application development, performance assessment, algorithms, and cloud technology.

DAVID AKOPIAN (Senior Member, IEEE) received the Ph.D. degree from the Tampere University of Technology, Finland. He is currently a Professor with The University of Texas at San Antonio (UTSA). Prior to joining UTSA, he was a Senior Research Engineer and a Specialist with Nokia Corporation. His current research interests include digital signal processing algorithms for communication and navigation receivers, positioning, dedicated hardware architectures, and platforms for software defined radio and communication technologies for healthcare applications. He is a fellow of U.S. National Academy of Inventors.

PALDEN LAMA received the B.Tech. degree in electronics and communication engineering from the Indian Institute of Technology and the Ph.D. degree in computer science from the University of Colorado Colorado Springs. Currently, he is an Assistant Professor with the Department of Computer Science, The University of Texas at San Antonio (UTSA). His research interests include cloud computing, autonomic resource management, and big data processing in the cloud.

PATRICIA CHALELA received the B.S. degree in community health from the Universidad Industrial de Santander, and the M.P.H. degree in health promotion/health communications and the D.P.H. degree in health promotion from the UT Health Science Center at Houston, TX, USA. She is currently an Associate Professor with the Institute for Health Promotion Research (IHPR), UT Health San Antonio (UTHSCSA), and an Associate Director for education and training programs. Her research interests include chronic disease prevention and control, particularly the role of epidemiological, environmental and individual psychosocial factors on health and disease, and racial/ethnic disparities with emphasis on Latino populations.

AMELIE G. RAMIREZ received the B.S. degree in psychology from the University of Houston, TX, USA, and the M.P.H. degree in health services administration and the D.P.H. degree in health promotion from the UT Health Science Center Houston, TX, USA. She is currently the Chair of the Department of Population Health Sciences and the Director and a Professor of the Salud America, IHPR, UTHSCSA. She is an internationally recognized Cancer Health Disparities Researcher and directs research on human and organizational communication to reduce chronic disease and cancer health disparities affecting Latinos, and more.