TigerAware Microservices: A Modern Backend for Improved Platform Scalability and Consistency

A Project

Presented to

The Faculty of the Graduate School

At the University of Missouri

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science, Computer Science

Implemented and Defended by

Connor Rowland

Prof. Yi Shang, Advisor

May 2020

Table of Contents

List of Figures v

List of Tables vi

Acknowledgments vii

1. Abstract 1

2. Introduction 2

2.1 Problem Description 4

2.2 Proposed Solution 6

3. Background and Related Works 8

3.1 MUDICL and TigerAware 8

3.2 Microservices 9

4. Existing TigerAware Architecture 12

4.1 Native Mobile Applications 12

4.2 Web Dashboard 13

4.3 Firebase Realtime Database 13

4.4 Distributed Single Tenancy 14

5. Dashboard Changes 15

5.1 Angular Frontend Framework 15

5.1.1 Improved Performance ... 16

5.1.2 TypeScript Support ... 16

5.2 Web Dashboard Hosting ... 17

5.3 Project Structure ... 18

5.4 Performance Improvement ... 20

6. Microservices ... 21

6.1 TigerAware Microservices ... 21

6.1.1 Benefits of Firebase Cloud Functions ... 21

6.1.2 Microservice API ... 23

7. Cloud Messaging ... 26

7.1 Hybrid Notifications ... 26

7.1.1 Local-Only Notifications ... 26

7.1.2 Remote-Only Notifications ... 27

7.1.3 Combining Local and Remote Notifications ... 28

7.2 Participant Messaging ... 35

8. Server Schedule Creation ... 38

8.1 Benefits of a Scheduling Microservice ... 38

8.1.1 Computing Active Days ... 42

8.2 Algorithm ... 42

9. Conclusion and Future Work 47

9.1 Future Work 47

10. Works Cited 49

List of Figures

Figure 1: Growth of annual funding for EMA studies 3

Figure 2: Existing TigerAware architecture 5

Figure 3: Proposed improved TigerAware architecture 7

Figure 4: Evolution of MUDICL EMA applications........ 9

Figure 5: Google search trends for microservices, Aug 2011 - Oct 2019 10

Figure 6: Microservice vs Monolithic application architecture [13]........ 11

Figure 7: Existing survey organization versus improved project hierarchy 19

Figure 8: Notification flow between participant devices and remote notification store........ 33

Figure 9: Firebase Cloud Messaging architectural overview [27] 34

Figure 10: Administrator messaging interface........ 36

Figure 11: Participant messaging interface........ 36

Figure 12: Notifications with overlapping compliance period 43

Figure 13: Example of failed notification scheduling........ 44

Figure 14: Improved random notification scheduling algorithm 45

Figure 15: Schedule resulting from the example in Figure 14 46

List of Tables

Table 1: TigerAware dashboard performance comparison 20

Table 2: TigerAware microservice API 23

Table 3: Comparison of local and remote notifications 29

Table 4: Hybrid notification store grammar 30

Table 5: Pending remote notification list grammar 31

Table 6: Shared message store grammar 37

Table 7: Definitions of schedule testing terms 40

Table 8: Schedule creation validation requirements 40

Table 9: Cases for computing survey active days 42

Table 10: Derived notification restrictions to guarantee schedule validity 46

Acknowledgments

First, I would like to thank my adviser, Dr. Yi Shang, for all his advice and guidance throughout my project and academic career. I would not have become the engineer I am today without his wisdom and generous support. I would also like to thank Dr. Tim Trull for his insight and recommendations during my project, which were an invaluable resource. Lastly, I would like to extend my gratitude to many wonderful people from the Mizzou Psychology Department – especially Dr. Tom Piasecki and Dr. Denis McCarthy – for the opportunities to work closely with them on their research projects.

Next, I would like to thank some of the many excellent people who I have worked with on the TigerAware platform over the last few years. Luke Guerdan and Will Morrison, who are fellow TigerAware co-founders, have worked closely with me on the platform since I started several years ago. I thank them not only for their leadership, engineering insight, and continued hard work on the TigerAware platform, but also for everything I have learned from working with them. I would also like to thank the other great engineers who have contributed significantly to TigerAware, including Zachary Kipping, Siyang Liu, Weiliang Xia, and Jayanth Kanugo.

Finally, I would like to thank my family for all their love and support. First, my parents, who have worked hard to provide me with every opportunity to succeed. I owe all my successes to them. Next, my brothers for all the laughs and continued friendship. Finally, my wonderful fiancée Regan. She continues to inspire me every day, and this project is dedicated to her.

1. Abstract

Smartphones have become an integral part of people's lives in the 21st century, and from social media to message boards smartphones have allowed a unique window into the lives of those around us. This can be especially useful for researchers conducting Ecological Momentary Assessments (EMA) in fields such as psychology or medicine. TigerAware is a unique mobile application that provides a flexible, customizable interface used by researchers to build EMA studies and deploy them to subjects. In this project, the tools provided in the existing TigerAware platform are improved by updating the backend technologies to a more modern microservice pattern. The existing technology stack of the TigerAware web dashboard utilizes a monolithic FEAN stack (Firebase, Express, AngularJS, and Node.js). Although this paradigm is suitable for smaller projects, it begins to suffer at scale due to a more complicated hosting process and fewer options for advanced integration with mobile platforms and other external services. Microservices are a more modern backend pattern which utilize many decoupled standalone services to perform business logic. By converting the existing stack to a microservice architecture, TigerAware can provide advanced functionality that improves quality of life for researchers and enables improved subject engagement. These improvements include upgrades to the TigerAware web dashboard, addition of cloud messaging capabilities, and improved notification scheduling.

2. Introduction

For many years, researchers across a large array of different disciplines have looked for ways to get insight into the lives of their subjects. With the proliferation of smartphones in our day-to-day life, researchers now have a valuable new tool they can utilize to this end. Smartphones have become massively prevalent in the 21st century – according to some of the most recent statistics, more than 81% of all Americans own a smartphone [1]. This number is even higher in younger populations, with at least 96% of young adults ages 18-29 owning smartphones. Additionally, modern smartphones come out of the box with an array of advanced sensors – GPS location services, internet and Bluetooth connectivity, and accelerometers just to name a few. These sensors can prove very useful to certain research projects, providing data and insight that was difficult or impossible to obtain in the past.

This readily available technology has popularized the Ecological Momentary Assessment (EMA), a type of study which seeks to capture the behaviors and experiences of subjects in real-time throughout their normal day-to-day lives [2]. EMA-based studies often rely on repeated random sampling to prompt users with a set of questions to answer at different times throughout the day. These studies have grown in popularity along with the availability of smartphones, and federal funding for EMA studies exceeded $165 million in 2019 [3]. Figure 1 shows the total annual funding for EMA studies since 2008. The data was collected from the National Institute of Health's Federal RePORTER, and includes projects funded by the NIH, Department of Veterans Affairs, National Science Foundation, and other federal agencies [3].

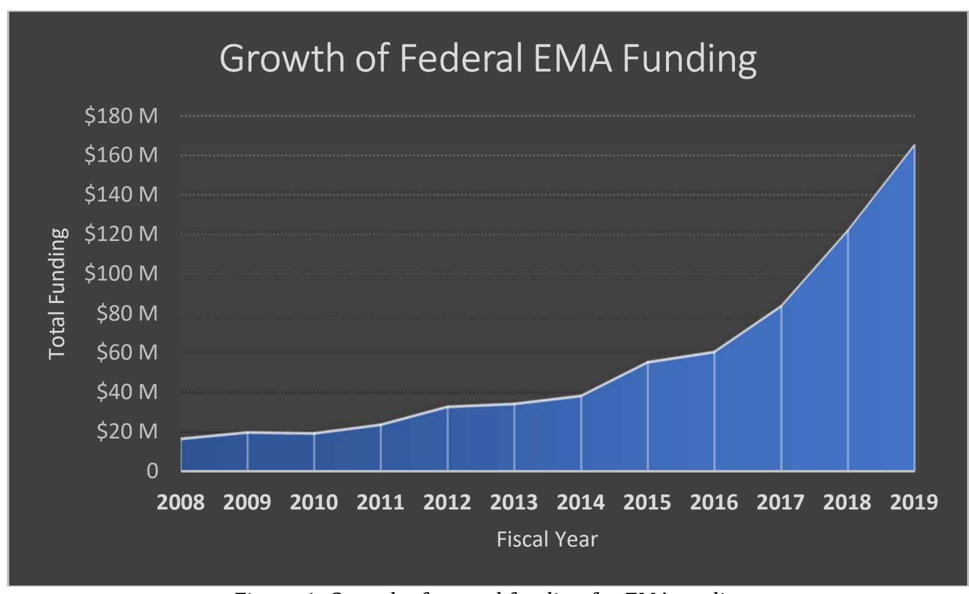

Figure 1: Growth of annual funding for EMA studies

Despite their growth in popularity, EMA studies can be difficult to design or administer given the high cost and long timeframe of paying a development team to write a custom mobile application for the study. Additionally, it is hard for a single 'out-of-the-box' application to cover the wide range of use cases encountered in different studies. For example, a researcher studying alcohol use among a college population may want focused bursts of prompts in the evening and the ability to interface with a blood alcohol sensor. A different researcher studying phantom limb pain in amputees may instead focus on prompts in the morning and user-initiated responses. To accommodate the innumerable possibilities in these studies, an EMA application must provide researchers with the flexibility and customizability to create a unique study protocol to suit their needs.

TigerAware is a new platform that allows researchers to design, build, and administer EMA studies to their participants. TigerAware consists of two major software components: native mobile applications which study participants use to receive prompts and respond to surveys, and a web dashboard which researchers use to build and manage their studies.

TigerAware includes many novel advancements that provide an edge over other software options in the EMA field. First, TigerAware is built on a highly flexible and customizable study framework. This allows researchers to choose from a wide variety of question types, survey flow controls, and user notification types. New question types, external sensor integrations, or other features can also be added to the existing platform with little time or financial investment.

2.1 Problem Description

TigerAware is an innovative platform that provides many new and exciting advancements over existing applications for EMA data collection. However, the existing technology stack used in the web dashboard and backend imposes limitations to platform scalability. The current TigerAware dashboard is built on a monolithic FEAN (Firebase, Express, AngularJS, Node.js) stack. This dashboard architecture requires dedicated hosting on either an in-house managed server or a dedicated hosting service such as Heroku. This creates issues with scalability in production since managing a monolithic web application requires a time-consuming deployment process and limits developer mobility towards adding new functionality or modifying existing functionality once the application is in production.

Currently, all the backend functionality of the TigerAware dashboard is housed within the monolithic web application. This poses several issues. First, functionality is hard to modify, and even a small change requires a rebuild and deploy of the entire web application to each customer. Also, code reuse is more difficult, which causes redundancies throughout the application. These redundancies cause problems if existing functionality needs to be

changed, since the same modification must be reproduced in multiple places in the codebase. A lack of segregated functionality also makes unit and integration testing harder to implement.

Finally, the existing backend architecture does not allow for advanced integration with the TigerAware mobile applications. Although the web applications are hosted for researcher use, there is no persistent backend service to allow for continuous communication between the 'brain' of the platform and mobile applications. This means that certain functionality that would ideally be housed centrally, such as participant schedule creation, must instead by implemented natively on each mobile application. This not only reduces platform consistency by requiring multiple separate implementations of the same algorithm, but also demands unnecessary computational load on the client-side applications. Also, advanced user engagement features, such as cloud messaging, cannot be implemented without a persistent backend service.

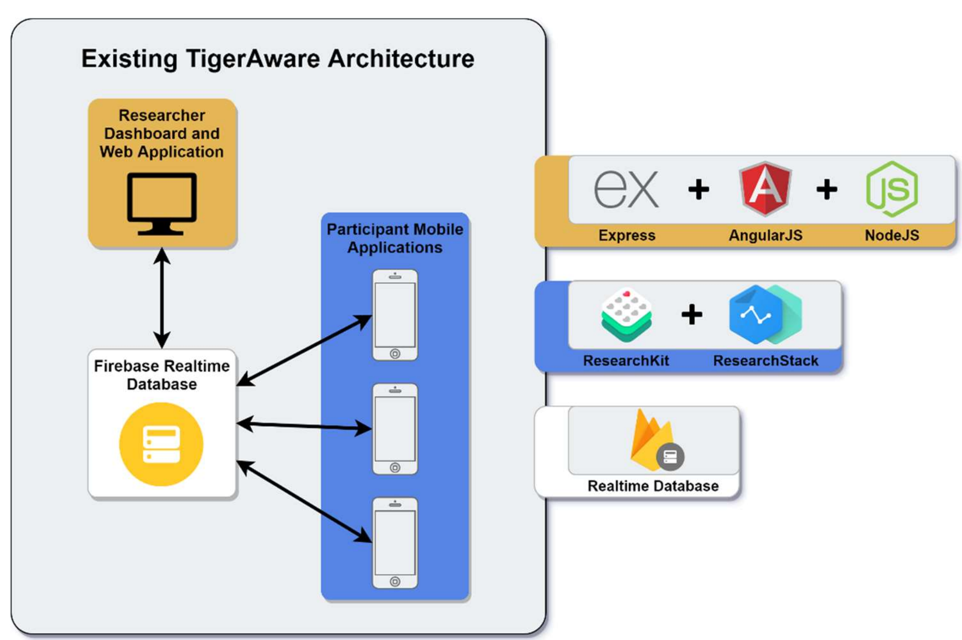

Figure 2: Existing TigerAware architecture

2.2 Proposed Solution

To improve the scalability of the TigerAware dashboard, the underlying technology stack is converted from the existing monolithic FEAN stack to an architecture utilizing Angular within a Firebase app. Since Angular has the option of being compiled into a single-page web application, rather than a full Node.js application, the new dashboard can easily be served using Firebase hosting. This solution enables a much more scalable end product, as well as a streamlined deploy process.

To improve the issues caused by the monolithic design of the existing TigerAware web application, the core functionality of the dashboard will be moved into segregated, self-contained microservices. This allows new or existing functionality to be modified and deployed easily, even in production. Rather than having to redeploy the entire web application each time a change is made, only a single microservice needs to be deployed. By reducing redundancies and encouraging code reuse, core functionality is more consistent and can be reasonably unit tested as part of the deployment pipeline.

To provide advanced integration with the TigerAware mobile applications, the newly developed microservices will be employed to migrate certain crucial business logic to the cloud and away from the mobile applications. First, the notification schedule creation logic is improved and implemented in a single, unified microservice. New user engagement features, such as cloud messaging, are also implemented using the new persistent backend services.

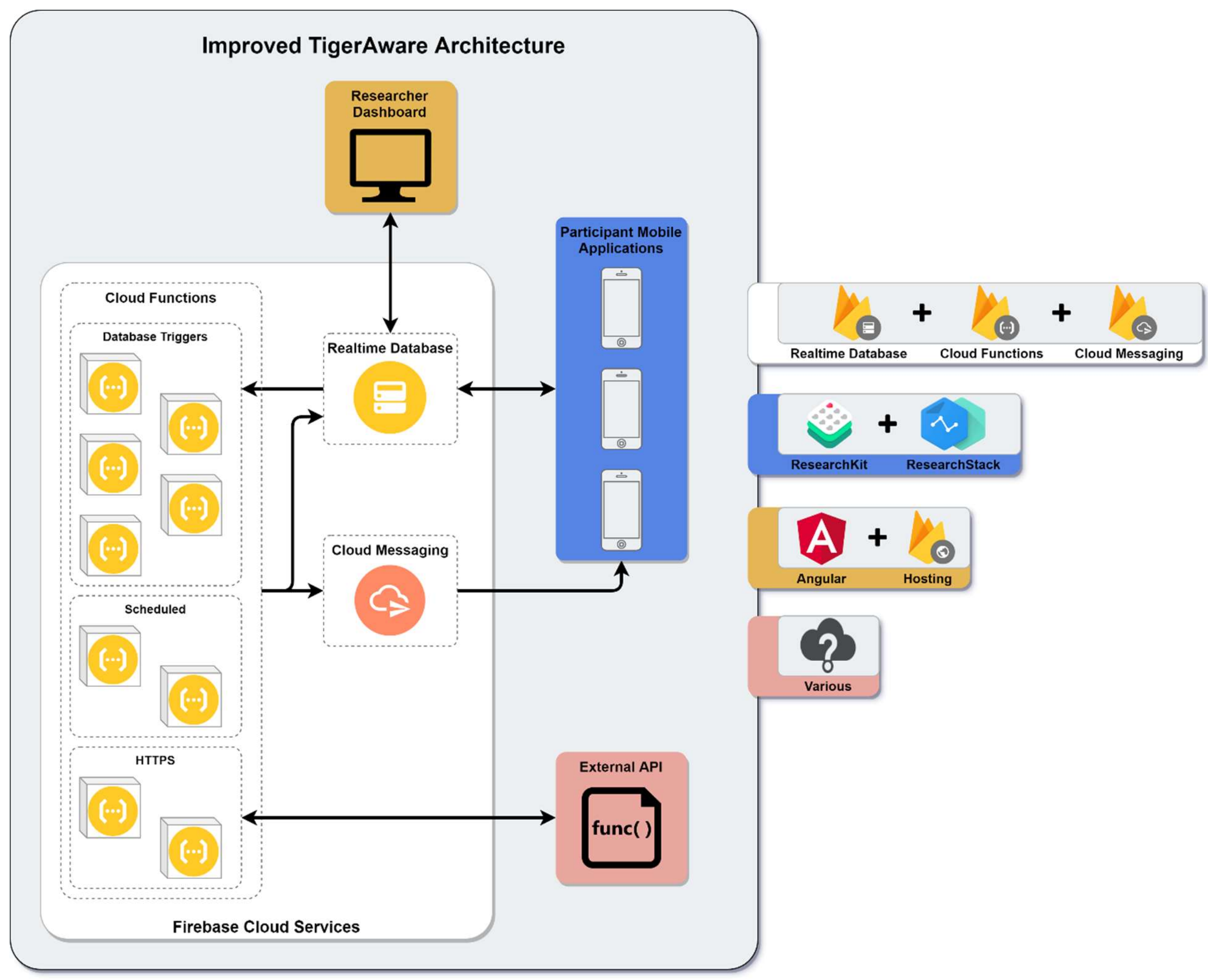

Figure 3: Proposed improved TigerAware architecture

3. Background and Related Works

For many years, researchers across a multitude of disciplines have sought out ways to study human participants in their daily lives. In the fields of psychology and medicine, Ecological Momentary Assessments (EMA) started to become the method of choice for researchers when introduced in 1994 [4]. However, this method has seen a spike in popularity due to the increasing availability of smartphones and other smart devices in recent years. The practical benefits of EMA studies were already being examined by psychology researchers at the University of Missouri as early as 2009 [5].

3.1 MUDICL and TigerAware

In the MU Distributed and Intelligent Computing Lab (MUDICL), demand grew for applications to aid researchers in completing EMA studies. In 2013, the lab completed an application to aid researchers studying alcohol craving through ambulatory assessment [6]. In 2015, the previous application was extended to provide new features for additional studies [7]. A few years later, a new web application was created for a psychological study of mood dysregulation [8]. Although these applications set the groundwork for EMA applications in the MUDICL, they were very inefficient – they were standalone applications containing features specific to a single psychological study.

Around this time a new group in the MUDICL, led by William Morrison, began work on a new project to create a single platform for EMA studies – modular and extensible enough that it could be applied to a wide range of different studies without the need to create a new application each time. This project would eventually become the current TigerAware

platform. Several reports were published about advancements and development work on the project, including work on the existing web dashboard [9] and development of the TigerAware Android application [10]. In 2018, an introduction and overview of the entire system was published [11].

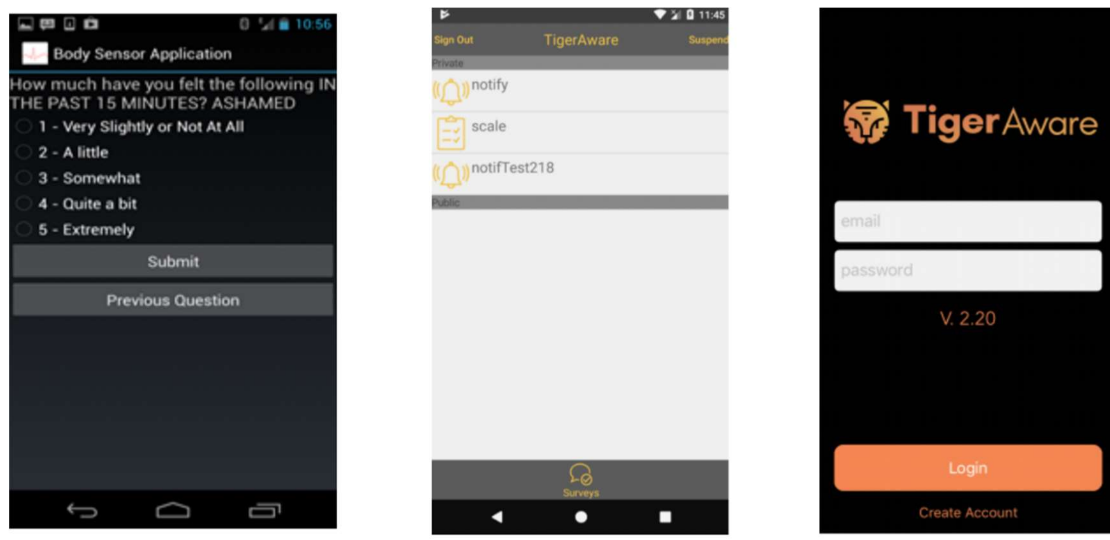

Figure 4: Evolution of MUDICL EMA applications from [6], [10], and the current TigerAware iOS application

3.2 Microservices

The microservice architecture is a relatively new paradigm in web applications, but one that is growing in popularity quickly. Many large companies have already migrated from monolithic applications to utilizing microservices including Amazon, Uber, and Netflix [12]. Figure 5 shows the growth of microservice popularity based on Google search trends between 2011 and 2019. The basis of microservice architectures is to write numerous small, standalone services rather than include business logic as part of one monolithic application [13]. The benefits of microservices are numerous, and include system resiliency, scalability, and fast development cycles.

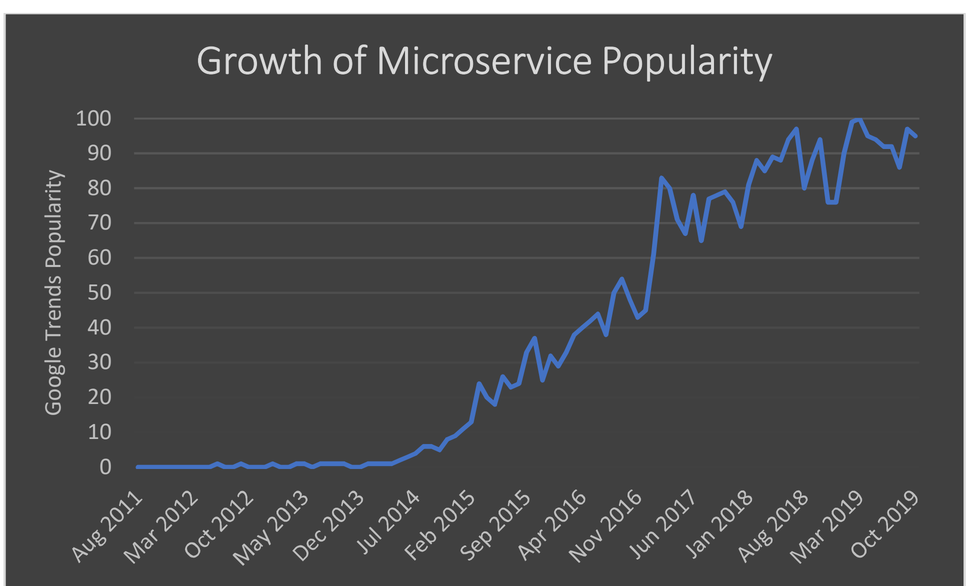

Figure 5: Google search trends for microservices, Aug 2011 - Oct 2019

The first major benefit of microservices is system resiliency. Since each microservice completes a single dedicated task, and is loosely coupled to other microservices, a failure in a single microservice will not cause a system-wide failure. This is a noticeable improvement over monolithic applications, where a failure anywhere in the system could render the entire application unusable [12]. Bugs and system failures are also easier to diagnose since developers can immediately narrow problems to a single microservice and identify exactly where errors are occurring.

Another benefit of microservices is system scalability. In a standard monolithic web application, such as the existing TigerAware dashboard, the entire application sits on a server and shares resources. Although load balancing and scaling can be made easier through a Platform as a Service (PaaS) such as Heroku or Amazon's EC2, the resources for the entire web application must be scaled up or down as a whole [22]. Microservices, on the other hand, can be load balanced and scaled individually as they are

needed [23]. Additionally, since most microservice providers spin up new instances of each service as they are invoked, microservices often require no manual scaling or partitioning from the developer at all.

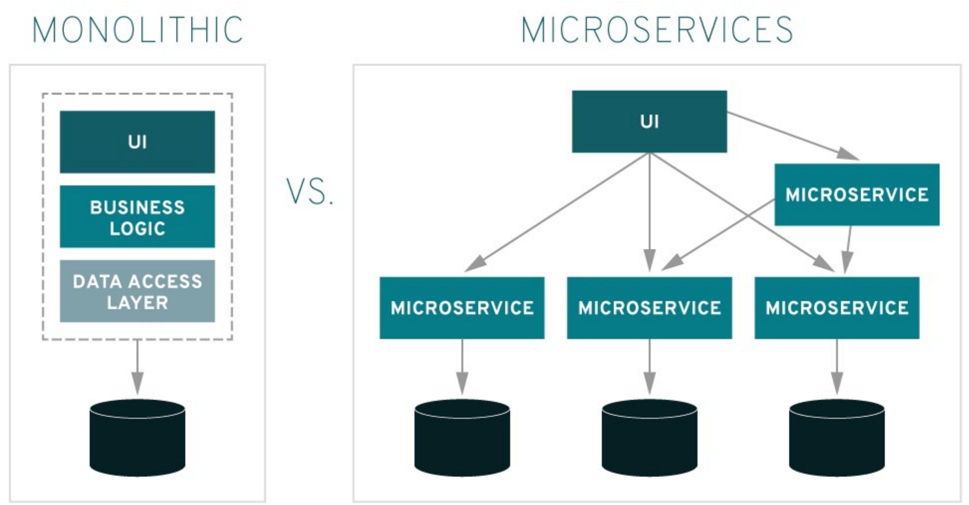

Figure 6: Microservice vs Monolithic application architecture [13]

The final major benefit of the microservice architecture is the increased speed of development cycles. Since microservices each service an atomic task independent of code outside the service, it is easy for developers to understand the workflow of a single microservice even if they are unfamiliar with the inner workings of other parts of the application [12]. This means that the addition of new functions or modification of existing functions is faster and easier than making changes to a monolithic application. Additionally, microservices can be deployed individually such that small or isolated changes only require single services to be redeployed rather than the entire application.

4. Existing TigerAware Architecture

The existing TigerAware architecture consists of three major portions: (1) native mobile applications on iOS and Android, which study participants use to receive prompts and respond to surveys, (2) a web dashboard which researchers and other administrators use to create, deploy, and manage their study, and (3) a Firebase database which serves as a datastore as well as real-time connection between platforms. For this report, the specific implementation of the mobile applications will not be covered in detail.

4.1 Native Mobile Applications

The TigerAware mobile applications are the interface survey participants use to interact with the surveys they are enrolled in. They consist of native applications written for both iOS and Android. Once participants have been enrolled in a survey by the survey administrator, they will be able to create a user account on the mobile platform of their choice. Then, each of their surveys will be downloaded to their device and their notification schedule will be created and stored locally. These notifications are scheduled directly with the operating system local scheduler.

The underlying survey flow structure of the applications utilize open-source libraries for building and displaying surveys – ResearchKit for iOS and ResearchStack for Android. These libraries provide SDKs for creating, building, and administering surveys as well as the UX framework for displaying them [14]. In addition to the survey SDK and UX framework, ResearchKit includes many 'Active Tasks' such as the Gait and Balance and Paced Serial

Addition Test [15]. These tasks can be quickly and easily integrated into the iOS application and be made available to researchers.

4.2 Web Dashboard

The TigerAware web dashboard is the interface that researchers and administrators use to build, deploy, and manage their surveys. It is built using Node.js, Express, and AngularJS, and hosted using Heroku. Once researchers have created a user account, they can use the dashboard's survey builder to create new surveys, add questions to their surveys, and add scheduled or random notifications to send to users. Once a new survey has been created, researchers can use the administration page to manage their participants, including adding new participants or removing existing participants. Also, survey responses can be visualized directly on the dashboard or the entire set of responses for a survey can be downloaded as a comma-separated values (CSV) file.

4.3 Firebase Realtime Database

Firebase's Realtime Database serves as the backend database for the TigerAware platform. Firebase is a NoSQL database created by Google which provides capabilities for real-time and offline functionality [16]. Their easy-to-use SDKs allow for simple integrations on both the web dashboard and mobile applications, which communicate with each other through the data stores in Firebase.

The TigerAware database schema consists of several high-level data stores: blueprints, compliance, data, and users. The blueprint store contains a list of surveys created by researchers. Blueprints are written to Firebase when a researcher creates a survey on the

web dashboard and are interpreted by the mobile applications to display surveys to participants. The compliance store contains records of user interaction with their surveys, including when notifications are received and when surveys are initiated or completed. The data store contains user responses for each survey, which can be visualized on the dashboard or downloaded as a CSV. Finally, the users store contains metadata about user accounts, including the list of surveys each user has created (in the case of researcher accounts), or the list of surveys in which each user is enrolled as a participant.

4.4 Distributed Single Tenancy

One important architectural feature of the TigerAware platform which sets it apart from many other applications is the need for a distributed, single-tenant system. TigerAware works with many different research groups from a variety of different institutions on a wide range of different research topics. Since many of these projects involve EMA-style studies with human subjects, data security is not only a top priority, but certain requirements must be met before TigerAware can even be selected as a vendor in most circumstances. To provide an additional layer of security for researchers and subjects alike, the TigerAware system is distributed to give each research group their own single-tenant system. Each research group has a private database, web dashboard, and mobile app configurations. Although this structure is great for security, it also brings unique development challenges; the technologies and design choices selected for TigerAware must be able to easily accommodate and facilitate a distributed system.

5. Dashboard Changes

The existing TigerAware dashboard consisted of a Node.js web application running Express and AngularJS. This architecture is great for quickly standing up web applications and can be served using a Platform as a Service (PaaS) provider such as Heroku. However, this paradigm can begin to suffer at scale, and it is difficult to transition into a distributed system if multitenancy is not desired. Additionally, several patterns on the existing TigerAware dashboard complicate researcher workflows. The addition of projects and other dashboard optimizations can help to streamline researcher workflows reduce unnecessary redundancies.

5.1 Angular Frontend Framework

The first improvement to the TigerAware web dashboard is upgrading the existing technology stack to more modern and scalable technologies. The current TigerAware dashboard operates on a FEAN (Firebase, Express, AngularJS, Node.js) stack. Although this is a popular stack used by many web applications, newer technologies can help to optimize scalability and performance. The new TigerAware dashboard will utilize Angular as a front-end framework and replace the existing Node.js and Express server with a microservice backend. By separating the frontend and backend of TigerAware's web application, performance and scalability can be improved.

Released in late 2016, Angular 2 (referred to as simply Angular) is the successor to Google's popular AngularJS web framework. Angular has many advantages over its predecessor, including improved performance, modularity, and TypeScript support [17].

5.1.1 Improved Performance

The new versions of Angular boast significantly improved performance over their old AngularJS counterparts. In some scenarios, the relative speedup can be as much as 5 – 10 *times* faster [17]. This performance increase is mainly due to more efficient binding checking in Angular. Rather than repeatedly checking each scoped variable for changes during a Digest Cycle, as was done in AngularJS, the new versions of Angular instead implement one-way change detection to update bindings. This change is thanks to the Observable paradigm – the most common asynchronous pattern used in frontend Angular applications. Observables are subjects that maintain a list of dependent observers who are relying on information about their internal state. When the state changes, the subject notifies each observer of the update [18]. This streamlines the binding process because observers no longer have to repeatedly check the subject for state changes.

5.1.2 TypeScript Support

One of the most important features of new Angular versions is support for TypeScript. TypeScript is a superset of standard JavaScript and provides many important improvements that aid in development. It is trans-piled to JavaScript when built, which means that it still benefits from all the optimizations of JavaScript and can run anywhere JavaScript can. TypeScript can still utilize existing standalone scripts or libraries written in JavaScript.

TypeScript also benefits from being statically typed and compiled. This allows developers to catch errors and bugs during compile time rather than runtime – speeding up development time and reducing bugs encountered in production. Strong typing also allows for type and interface definitions. The new TigerAware dashboard makes extensive use of

type and interface definitions to enforce types throughout the application. This process helps to enforce patterns and style throughout the application as well as improve code reusability.

5.2 Web Dashboard Hosting

In addition to improving the efficiency of the TigerAware dashboard's front-end framework, the hosting strategy was also improved. The existing TigerAware web application was hosted using Node.js dynos on Heroku, a PaaS hosting provider with support for many different application environments. Although Heroku is relatively easy to get started, there are several notable drawbacks: distributed applications are difficult to manage, automated deployments can be complex, and standard dynos become expensive with multiple hosted applications. These issues can be avoided by switching the TigerAware Dashboard to Firebase Hosting.

The first major issue with hosting the TigerAware web application on Heroku is the difficulty in managing distributed applications. As discussed in the architecture overview, the TigerAware system requires a distributed, single-tenant architecture to enhance platform security. However, this type of paradigm is difficult to manage using Heroku. Each web application must be created and assigned its own dyno(s). Then, every time a new update is merged into the application, it must be configured and deployed to each Heroku app separately. The requirement of deploying to Heroku apps through git makes this process difficult to automate, and manual configurations and deployments are time-consuming and error-prone.

Firebase Hosting, on the other hand, makes it much easier to work with multiple applications in a distributed pattern. Firebase configurations are easy to swap in and out, and the Firebase CLI makes deploying to apps simple [19]. Deploying to multiple different hosted applications with different configurations is straightforward and can be automated with a short bash script (around 25 lines of code).

Heroku applications can also become expensive with multiple hosted applications. The smallest standard dyno (non-hobbyist) currently offered by Heroku costs $25 per month – regardless of traffic or usage [20]. With the number of projects currently supported by TigerAware, hosting one web application for each research group would cost at least $3,000 annually, not including any additional cost for scaling to meet high-traffic periods. Firebase Hosting, however, does not base prices on different tiers of resource allocation. Instead, prices are fixed based on usage, and sites must cross the free-tier usage thresholds before any costs are incurred [21]. Based on the current TigerAware dashboard usage, the free tier usage is not yet being exceeded each month – meaning the hosting for TigerAware's web dashboard is free using Firebase Hosting.

5.3 Project Structure

The final modification to TigerAware's web dashboard is the addition of an abstracted project structure. In most research protocols that TigerAware works with, investigators want to deploy a set of several surveys to users. Although this is possible with the existing dashboard, the workflow for managing several surveys in the same protocol is obtuse and not streamlined. When new users are added to the protocol, a researcher must go through each of the surveys individually to add the user. Downloading survey responses is also only

possible on an individual survey level. This means that data for multiple surveys is downloaded in separate CSV files and researchers must manually merge the files to view data for the entire protocol.

To streamline the researcher workflow when managing protocols with numerous surveys, the new TigerAware dashboard includes an abstracted grouping mechanism for surveys called projects. A project denotes a group of surveys which researchers would like to share a common list of administrators and participants. Multiple individual surveys can be added to the project, and when a participant is added to the project, they are automatically enrolled in each of the individual surveys. Additionally, survey data can be downloaded for every survey in a project at the same time, saving researchers valuable time when analyzing their data.

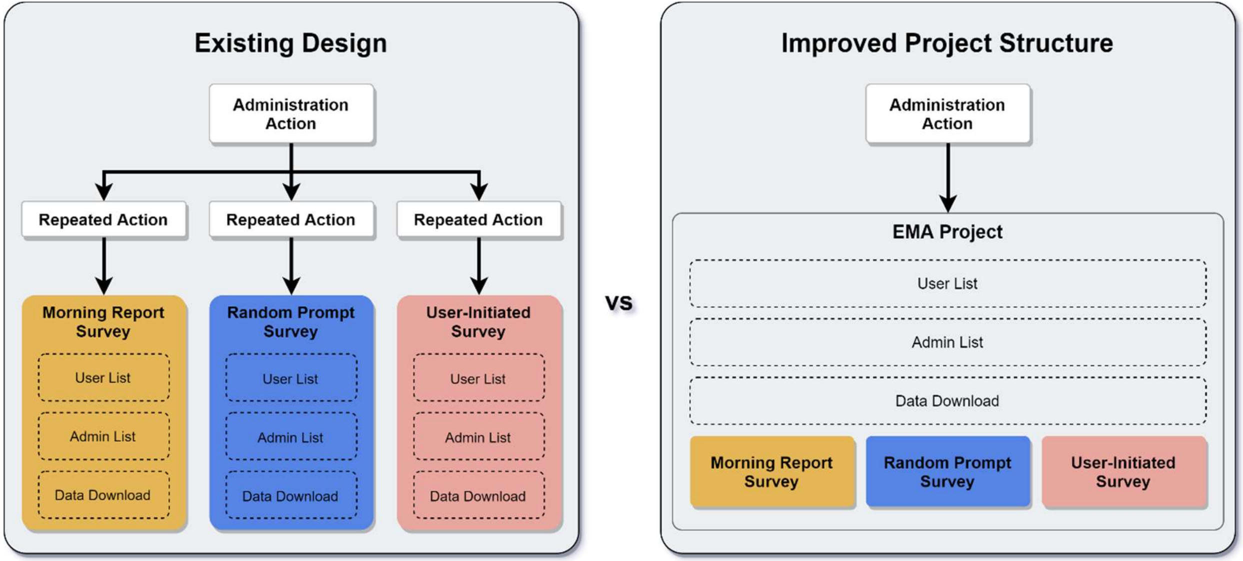

Figure 7: Existing survey organization versus improved project hierarchy

5.4 Performance Improvement

Following is a performance comparison between the existing TigerAware dashboard and improved Angular implementation. These results include performance improvements provided by upgrading the dashboard to Angular, migrating from Heroku to Firebase Hosting, and improved querying enabled by the project structure. The tests were conducted using a cleared cache and a throttled 3G network connection to ensure a fair comparison.

Table 1: TigerAware dashboard performance comparison

	Login Page Load Time	**Overview Page Load Time**	**Relative Performance Increase**
Existing TigerAware Dashboard	8.92 sec	8.83 sec	-
Improved TigerAware Dashboard	1.80 sec	2.39 sec	**423.6%**

6. Microservices

The existing TigerAware backend was implemented as part of the monolithic Node.js web application. Any backend workflow was completed through an Express API built into the existing Node application. This paradigm not only increased the size and complexity of the web application, but also required hosting on a persistent server environment. To improve the flexibility, modularity, and scalability of the TigerAware backend, the existing Express API is reworked into a brand-new microservice architecture hosted on Firebase Cloud Functions. This microservice structure interfaces with the updated frontend web dashboard.

6.1 TigerAware Microservices

TigerAware's microservice backend is designed to handle most business logic from the web dashboard frontend as well as expose endpoints accessible directly from the mobile applications. Each microservice is implemented using TypeScript and deployed on Google Cloud Functions.

6.1.1 Benefits of Firebase Cloud Functions

Firebase Cloud Functions are a service provided by Google to easily host remotely executable code in the cloud. Cloud Functions automatically scale to meet demand without any manual provisioning needed by developers [22]. This frees up developer time for writing new features or maintaining existing code rather than dealing with scaling backend resources. Cloud Functions are also priced based only on usage and include a free tier. The first 125K function invocations are free each month, and invocations are only $0.40 per

million afterward. These features allow TigerAware to host persistent microservices with minimal overhead, time investment, and cost.

In addition to the managerial benefits, Cloud Functions also provide implementation options that help to optimize backend performance. Functions have several options for invocation, including HTTPS as well as Firebase triggers [22]. Firebase triggers are especially useful since they allow for microservices to be automatically executed when database records are modified. For example, a function that updates notification schedules can be automatically executed when a new schedule is written into Firebase. This workflow allows TigerAware's microservices to automatically perform maintenance functions and reference updates without ever needing to be invoked directly. Specific functions can also be invoked directly over HTTPS or configured to run on a schedule.

The final benefit of Firebase Cloud Functions is the integrated authorization environment that function invocations are executed in. Since Firebase Cloud Functions are packaged and deployed to a specific Firebase instance, they are automatically configured to be triggered by, and reference, their parent instance [22]. This removes the need to provision or configure microservices to a specific database. This is especially important for TigerAware's distributed single-tenant system; a unified set of microservice code can be easily deployed to any number of distributed systems without any configuration swapping. Each service will automatically be connected to its own system.

6.1.2 Microservice API

TigerAware's microservices utilize the following invocation types:

- Firebase Write: triggers when data is created, updated, or deleted in Firebase
- Firebase Create: triggers when new data is created in Firebase
- Firebase Update: triggers when existing data is updated in Firebase
- Firebase Delete: triggers when data is deleted from Firebase
- HTTPS: triggers when an HTTPS endpoint is requested. Handles GET, POST, PUT, DELETE, and OPTIONS requests
- Scheduled: triggers on a predefined cron-like schedule

Table 2: TigerAware microservice API

Function	Invocation	Description
activeParticipantUpdate	Firebase Update	Adds references to user's taking lists when they are added to a new project
addAdminToProject	Firebase Create	Updates survey administrators and user's survey list when they are added as a project administrator
addParticipantToProject	Firebase Create	Updates survey participant list and user's taking list when they are added as a project participant
addUserByLink	HTTPS	Adds participant to project taking list and updates link metadata when a participant is added via invite link
cloneProjectForUser	HTTPS	Creates a clone of an existing project

complianceResponseCreate	Firebase Create	Cancels the next 24 hours of a user's remote notifications when a new compliance event is created
downloadSurveyData	HTTPS	Downloads survey responses for a project or individual survey. Can be invoked by researchers with API key
getProjectMetadata	HTTPS	Returns certain project metadata for unauthenticated users
notificationDelete	Firebase Delete	Removes a pending remote notification when the originating notification is removed from the schedule
notificationScheduleCreate	Firebase Create	Schedules a pending remote notification when a new originating notification is added to the schedule
notificationScheduleUpdate	Firebase Update	Modifies a pending remote notification when the originating notification is updated
notificationSoundUpdate	Firebase Write	Updates the alert sound on notifications when a new sound is selected on the project
offsetsUpdate	Firebase Update	Updates notification offsets on each survey when the project offsets are modified
projectDelete	Firebase Delete	Removes references for administrators, participants, and surveys when a project is deleted

projectDurationUpdate	Firebase Update	Updates duration on each survey when the project duration is modified
removeAdminFromProject	Firebase Delete	Updates survey administrator list and user survey list when they are removed as a project administrator
removeParticipantFromProject	Firebase Delete	Updates survey participant list and user taking list when they are removed as a project participant
retrieveNotificationsToSend	Scheduled	Polls the pending notification list to determine which hybrid notifications should be delivered
sendHybridNotification	HTTPS	Sends a hybrid notification to a participant given the notification contents and messaging tokens
sendMessagingNotification	Firebase Create	Sends an admin messaging notification to a participant given the notification contents and messaging tokens
surveyCreate	Firebase Create	Updates project admin and participant lists when a new survey is created
surveyDelete	Firebase Delete	Updates project admin and participant lists when a survey is deleted
takingUnschedule	Firebase Delete	Removes pending hybrid notifications when a user is removed from a survey
validateLink	HTTPS	Determines if a given invite link is valid

7. Cloud Messaging

In many EMA studies, researchers are interested in ways to interact with participants remotely while they are in the field. Whether notifying participants to take surveys, checking in with compliance, or alerting participants about updates with the study, it would be convenient for TigerAware to include a set of built-in tools for participant engagement. The existing TigerAware platform relies solely on local mobile notifications. Although local notifications can have a higher rate of delivery compared to remote notifications, there are several advantages to utilizing remote notifications in addition to local notifications.

7.1 Hybrid Notifications

The first major benefit of adding remote notification capabilities to the TigerAware platform is the ability to utilize a hybrid notification delivery scheme. Most study protocols that TigerAware works with are based around a set of notifications that alert participants to respond to different surveys throughout the day. The original TigerAware architecture utilizes local, on-device alerts to deliver these notifications. Local notifications have the benefit of high deliverability; they don't require an active internet connection and are delivered as long as the participant's mobile device is powered on [23].

7.1.1 Local-Only Notifications

Despite the high rate of deliverability, however, there are drawbacks to using only local notifications. The most notable among these is that the iOS operating system enforces a limit of 64 locally-scheduled notifications per application [24]. Although this limit would

be fine for most applications, TigerAware often deals with protocols with many notifications per day. The following is an example of a common protocol structure:

- Morning Report: 1 notification per day, 2 reminders per notification
- Random Prompts: 3 notifications per day, 2 reminders per notification
- Evening Report: 1 notification per day, 2 reminders per notification

In total, this example protocol would use 15 notifications per day – local notifications would run out in just four days. However, most protocols run for several weeks at a minimum, and notifications need to be scheduled for the entire duration. To handle this limitation (and protocols with even more notifications) the current TigerAware applications schedule only the next two days' notifications at a time. This handles protocols with up to 30 notifications per day and works fine as long as participants open the TigerAware application at least once every two days. However, after two days without opening the application, the notifications stop. This poses serious issues for EMA protocols since the scientific integrity of the study relies on user prompts being accurate and consistent.

7.1.2 Remote-Only Notifications

The most obvious solution to the local notification limit is also the simplest: send all of TigerAware's notifications from the cloud. A server is always running, has no notification limit, and could send notifications to all TigerAware users from a unified service. This is a good solution, but still encounters the issue of notification deliverability. Remote notifications rely on user devices having strong network connections every time a notification needs to be delivered. Most applications can rely on notification retry mechanisms; even if a notification cannot be delivered *immediately*, it will be delivered

eventually. TigerAware, however, works with EMA protocols with very specific notification schedules which affect the scientific findings of studies. If these notifications can't be delivered immediately, they often shouldn't be delivered at all. This doubles down on the issues of notification deliverability, and significantly increases the number of missed notifications.

Another factor that can reduce the consistency of remote notifications to participant devices is device-specific delivery factors. These can vary greatly in different operating systems and devices. One common factor which can affect remote push notification delivery is device battery conservation. Both iOS and Android, in an attempt to save battery life, delay standard priority notifications when the device is in a sleeping/dozing mode [25]. Also, applications typically have a maximum quota of 'high priority' messages they are allowed to deliver within a short timeframe. Although all of TigerAware's push notifications are high priority (immediate delivery is necessary), the system may become locked out of sending notifications for protocols with a high volume of prompts. Finally, devices have different standards for push notification delivery based on application background permissions. Overall, there are many different factors which not only reduce remote notification deliverability, but also make them hard to predict consistently.

7.1.3 Combining Local and Remote Notifications

The best option for balancing notification persistence and consistency is a scheme that combines local and remote notifications. Local notifications should be prioritized when possible, but remote notifications can pick up the slack once local notifications run out. This

approach combines the best of both worlds; as many notifications as possible have the high consistency of local delivery, but it isn't possible to ever run out of notifications.

Table 3: Comparison of local and remote notifications

	High Deliverability (first 2 days)	**Supports Offline Participants**	**Avoids 2-Day Notification Limit**	**Allows Off-Device Scheduling**
Local-Only Notifications	✓	✓	✗	✗
Remote-Only Notifications	✗	✗	✓	✓
Hybrid Notifications	✓	✓	✓	✓

The first change that needs to be added to the existing TigerAware platform to facilitate hybrid notification delivery is a unified notification store shared between mobile applications and the TigerAware dashboard. Notification schedules can also be created directly in the cloud, which will be discussed in a further section. The shared notification schedule must store all the information required for remote delivery: participant id, time of desired delivery, survey blueprint key, notification compliance duration, and the platform which is currently assigned to handle the delivery.

Table 4: Hybrid notification store grammar

⟨notification-store⟩	⟨user-id⟩ : { ⟨notification-list⟩ } ⟨user-id⟩ : { ⟨notification-list⟩ }, ⟨notification-store⟩
⟨notification-list⟩	⟨notification-id⟩ : ⟨hybrid-notification⟩ ⟨notification-id⟩ : ⟨hybrid-notification⟩, ⟨notification-list⟩

⟨hybrid-notification⟩	{
	⟨blueprint-id⟩,
	⟨human-date⟩,
	⟨compliance-duration⟩,
	⟨notification-id⟩,
	⟨handling-platform⟩,
	⟨unix-time⟩
	}

⟨user-id⟩	*string*
⟨notification-id⟩	*string*
⟨blueprint-id⟩	‘bpid’ : *string*
⟨human-date⟩	‘date’ : *string*
⟨compliance-duration⟩	‘duration’ : *number*
⟨notification-id⟩	‘nid’ : *string*
⟨handling-platform⟩	‘platform’ : ‘mobile’ ‘platform’ : ‘unhandled’
⟨unix-time⟩	‘time’ : *number*

The main synchronization between the local and remote notification systems occurs through the shared notification store. The ⟨handling-platform⟩ attribute on each notification denotes whether or not the notifications have already been scheduled locally by the TigerAware mobile application. When a participant's schedule is first created, every notification will default to remote delivery. Then, each time the mobile application is opened and connects to Firebase, it 'claims' two days' worth of notifications by changing the ⟨handling-platform⟩ to 'mobile'. This change will kick off a chain of microservices in real-time to synchronize the local and remote notifications.

Once a user's notification schedule is written to the shared notification store, the *notificationScheduleCreate* microservice is automatically triggered with the new schedule. For each notification in the new schedule, this service will check the ⟨handling-platform⟩. If the platform is marked as 'mobile', the server can ignore the notification since it is already being scheduled on-device. If the platform is marked as ⟨unhandled⟩, then the notification should be prepared for remote delivery. The service will transform the notification into the correct format and push it onto the global list of pending remote notifications.

Table 5: Pending remote notification list grammar

⟨pending-notifications⟩	⟨notification-id⟩ : ⟨delivery-info⟩ ⟨notification-id⟩ : ⟨delivery-info⟩, ⟨pending-notifications⟩
⟨delivery-info⟩	{ ⟨blueprint-id⟩, ⟨compliance-duration⟩, ⟨notification-id⟩, ⟨participant-id⟩, ⟨unix-time⟩

	}
⟨notification-id⟩	*string*
⟨blueprint-id⟩	'bpid' : *string*
⟨compliance-duration⟩	'duration' : *number*
⟨notification-id⟩	'nid' : *string*
⟨participant-id⟩	'participantId' : *string*
⟨unix-time⟩	'time' : *number*

The global pending notification list stores a master list of all future notifications which are currently expected to be delivered remotely. That is, all the future notifications for every user which is not already scheduled directly on their mobile device. Two microservices ensure that this list is always up-to-date: *notificationDelete* and *notificationScheduleUpdate*.

The n*otificationDelete* service is triggered when an originating notification is deleted from a user's notification store. Notifications can be deleted from the notification store in a few situations, including if a user is removed from a protocol or if a researcher wants to reset their study participation. The *notificationDelete* service ensures these deleted notifications are also removed from the pending list so they will not be delivered.

The *notificationScheduleUpdate* service is triggered when an originating notification is modified. If the notification was modified to change the ⟨handling-platform⟩ to 'mobile' to

denote that it has been scheduled on-device, then the service will remove the notification from the list of pending notifications to ensure it is not delivered twice. If the ⟨handling-platform⟩ is still ‘unhandled’, the service will update the relevant ⟨delivery-info⟩ fields on the pending notification.

The final service which interacts with the pending notification list is the *retrieveNotificationsToSend* service. This survey runs automatically every 60 seconds and queries the master pending notification list to determine which hybrid notifications, if any, need to be delivered remotely. To perform this query efficiently, the pending notification list is stored with a secondary index on the ⟨unix-time⟩ field of notification. When a notification is found that needs to be delivered, the final step in the process is to invoke the *sendHybridNotification* service.

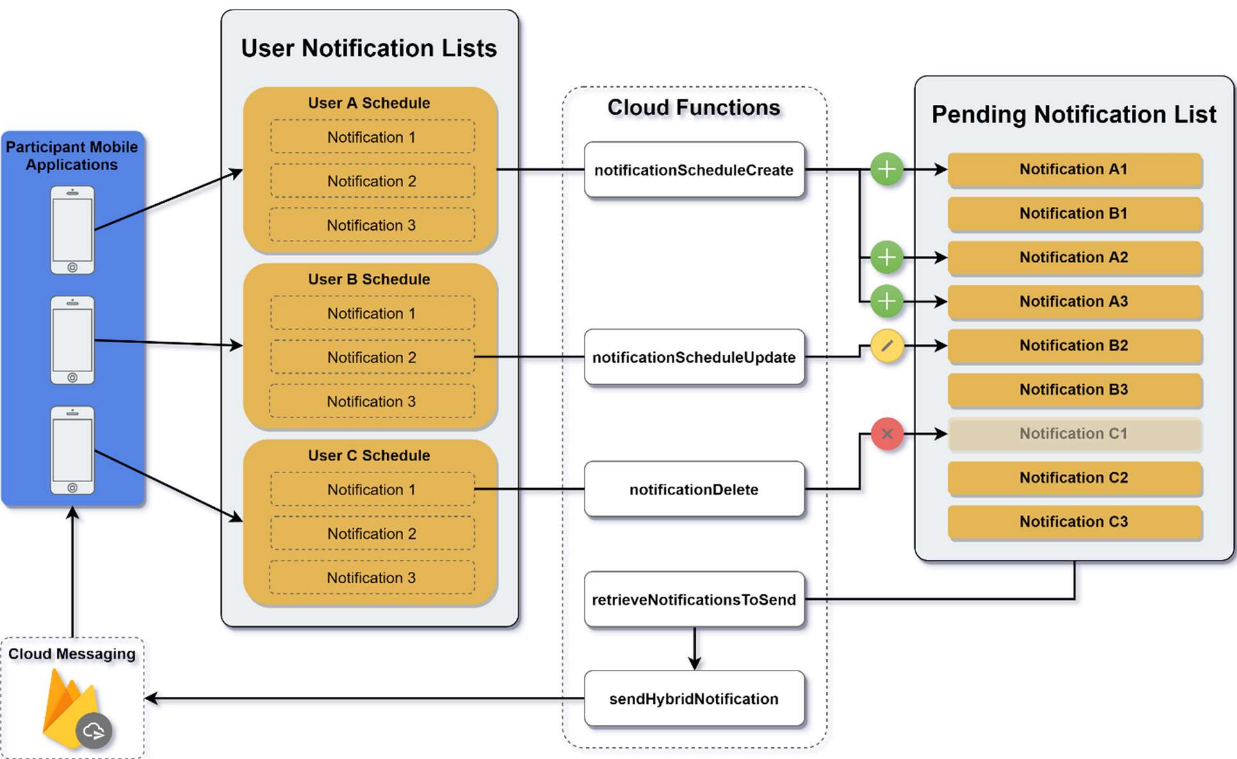

Figure 8: Notification flow between participant devices and remote notification store

The *sendHybridNotification* service is in charge of routing and delivering hybrid notifications to participants given the participant's messaging token and the notification's parameters. TigerAware utilizes Firebase Cloud Messaging to deliver push notifications. FCM is a great choice for notification delivery because it can handle cross-platform delivery to both iOS and Android [26]. Messaging to Android devices can be delivered directly, and iOS messages are passed through Apple Push Notification Service (APNs) automatically. When sending remote push notifications, the *sendHybridNotification* service includes appropriate survey information, such as the survey title and notification message, to ensure that remote notifications appear identically as local notifications to users. This allows hybrid notification delivery to appear seamlessly to participants, and they often won't be able to distinguish when the system has transitioned from local to remote delivery.

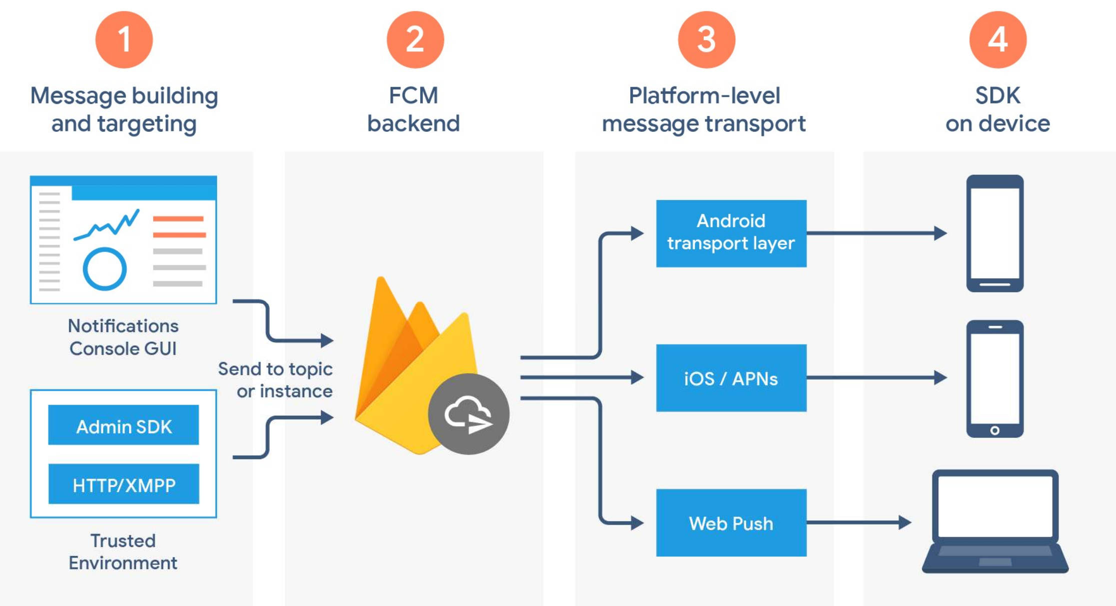

Figure 9: Firebase Cloud Messaging architectural overview [27]

The default behavior of FCM includes automatic retry for push notifications which cannot be delivered immediately. If a device is offline or otherwise unavailable when a

notification is sent, the notification will be stored and delivered when the device is next available, up to 28 days later [25]. Although this is a helpful feature for many applications, message retry is not desired for TigerAware notifications. Since each notification is linked to an EMA prompt, which is very time-sensitive, TigerAware notifications need to be volatile; if they can't be delivered immediately when sent, they should be discarded and never delivered. FCM provides options for creating volatile push notifications for both Android and iOS.

7.2 Participant Messaging

A second benefit of adding remote notification capabilities to the TigerAware platform is the option for increased user engagement through participant messaging. Throughout an EMA study, there are many times that researchers need to contact participants in the field – to update them on the status of their participation, to troubleshoot issues, or to notify them of modifications to the study. It would be convenient for researchers to have a built-in interface directly on the TigerAware platform for communicating with their participants.

Through the addition of cloud messaging capabilities, it is possible to add a participant messaging interface to the project administration dashboard on the TigerAware dashboard. This interface allows researchers to select any participant in their study, view previous messages with the participant, and send new messages to the participant in real-time. Although the participant will not be able to respond to messages directly from the TigerAware app, they will receive a push notification alerting them that they have received

a new message from the study administrator. They will then be able to log into a participant-facing version of the TigerAware dashboard to view and respond to their messages.

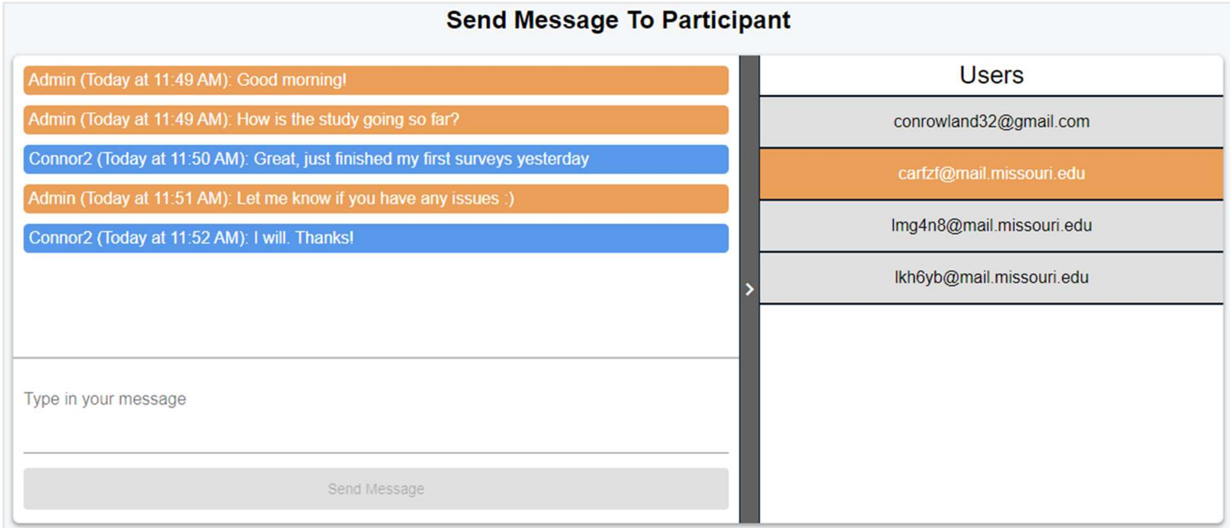

Figure 10: Administrator messaging interface

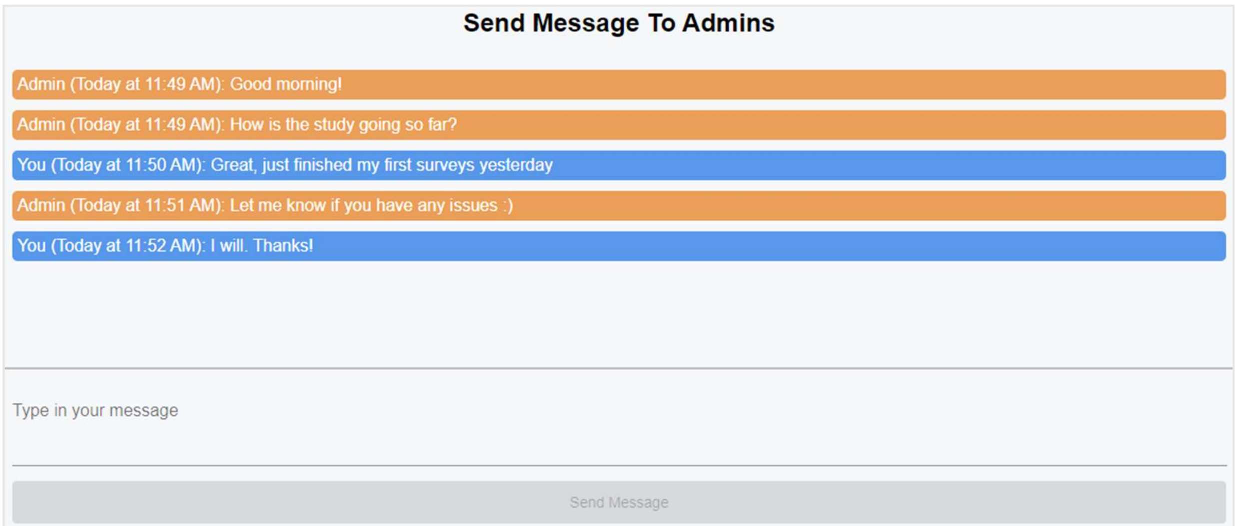

Figure 11: Participant messaging interface

Once an administrator sends a message to a participant, the data is stored in a shared messages store. This store is queried to display message history between administrators and each participant and will also automatically trigger the *sendMessageNotification* microservice when new messages are written. This service will pull the relevant messaging

information from the shared store and send the participant a push notification through Firebase Cloud Messaging.

Table 6: Shared message store grammar

⟨messages-store⟩	⟨project-id⟩ : { ⟨participant-list⟩ } ⟨project-id⟩ : { ⟨participant-list⟩ }, ⟨messages-store⟩
⟨participant-list⟩	⟨participant-id⟩ : { ⟨messages-list⟩ } ⟨participant-id⟩ : { ⟨messages-list⟩ }, ⟨participant-list⟩
⟨messages-list⟩	⟨message-id⟩ : { ⟨message⟩ } ⟨message-id⟩ : { ⟨message⟩ }, ⟨messages-list⟩

⟨message⟩	{
	⟨message-body⟩,
	⟨from-id⟩,
	⟨unix-time⟩
	}

⟨project-id⟩	*string*
⟨participant-id⟩	*string*
⟨message-id⟩	*string*
⟨message-body⟩	'body' : *string*
⟨from-id⟩	'from' : *string*
⟨unix-time⟩	'timeStamp' : *number*

8. Server Schedule Creation

In most EMA studies, prompted surveys create the protocol structure and direct user interaction with the study. To deliver scheduled and random prompts throughout the duration of a study, TigerAware needs to build and store a unique notification schedule for each participant when they begin a protocol. This process cannot be completed ahead of time because the start date for each participant is unknown until they start the study. The current TigerAware system creates schedules directly on the mobile application when participants are enrolled in a study. However, the process of schedule creation can be improved by migrating to the newly created TigerAware microservices.

8.1 Benefits of a Scheduling Microservice

There are several benefits to migrating schedule creation to the cloud. First, it allows all schedule creation to be completed in a single, unified service. Currently, each TigerAware mobile application has its own version of the participant scheduling algorithm. They are written in two different languages and hosted in separate repositories. This can be problematic because although the scheduling algorithm should work identically on both platforms, it is difficult to verify that this is the case in all scenarios. Also, whenever a change or new feature needs to be added to TigerAware's schedule creation, the change has to be replicated on both platforms. This not only takes extra development hours to implement and test but can potentially introduce discrepancies between the systems.

Migrating schedule creation to the cloud helps to improve system consistency by providing a single, unified service available to both mobile platforms. When a participant

first signs into a new protocol, the mobile applications invoke the scheduling service over HTTPS. The participant's schedule for the entire protocol is then created remotely and stored in the shared notification store (see Table 4). As before, this will automatically trigger the necessary services to keep remote delivery up to date.

The second benefit of migrating scheduling functionality to the cloud is improved testability. The current mobile implementations for scheduling are not only segregated, but also difficult to test due to deep integration with the rest of the TigerAware mobile application code. Adding these functions to a continuous integration (CI) pipeline would require significant changes to the existing codebases.

Microservices, on the other hand, allow for much easier testing due to each service already being standalone. Firebase provides a testing SDK which can easily integrate to provide unit testing for existing Cloud Functions [28]. The Firebase Test SDK utilizes Mocha, an open-source JavaScript testing framework, to run individual tests. Mocha allows testing of asynchronous microservices with an expansive suite of features, including support for both synchronous and asynchronous functions, automatic retries, and dynamically generated tests [29]. These features allow unit tests to be easily written and applied to all of TigerAware's microservices, including the schedule generation functionality.

First, a set of 'standard' projects with survey blueprints are created. These surveys should cover each of the possible notification functionalities provided by the TigerAware survey builder including different notification counts, compliance durations, and reminders. Once these projects are created, they can be stored in a development Firebase database just as real projects would; the Firebase Test SDK includes 'online' testing functionality which

directly interacts with real data in Firebase [28]. This is especially useful as new testing data can easily be added directly from the TigerAware dashboard when new notification features are added.

Once the testing data has been added to Firebase, tests are written to ensure that scheduling is working correctly. Since schedule creation is a stochastic process, there is rarely a single 'correct' schedule for each survey blueprint. Rather, there is a set of assertions that must be satisfied for a schedule to be considered 'valid'.

Table 7: Definitions of schedule testing terms

Term	Definition
Survey Active Day	A survey active day is a day in which the corresponding survey has at least one scheduled prompt (see Computing Active Days).
Project Active Day	A project active day is a day in which at least one of the corresponding project's surveys is active.
Daily Prompt Notification Count	The daily prompt notification count is the number of notifications in a certain active day for a specific prompt. This value is (*number of originating notifications* * *number of reminders*).

Table 8: Schedule creation validation requirements

Assertion	Explanation
Project Schedule Duration	The total duration of notifications for a project's schedule should match the difference between the first active day of the project and the last active day of the project.

Survey Schedule Duration	The total duration of notifications corresponding to a survey should match the difference between the first active day of the survey and the last active day of the survey.
Project Schedule Days	The total number of days of notifications for a project's schedule should match the number of active project days.
Survey Schedule Days	The total number of days of notifications corresponding to a survey should match the number of active survey days.
Daily Survey Notification Count	The number of notifications corresponding to a survey on a specific active day should match the sum of daily prompt notification counts for all prompts in the survey.
Daily Project Notification Count	The number of notifications in a project's schedule on a specific active day should match the sum of daily survey notification count for all surveys in the project.
Scheduled Notification Time	The time for a scheduled prompt should match the blueprint time, offset by the participant's time zone, on each of the survey's active days.
Random Notification Time	The time for a random prompt should be a time in between the blueprint start and end time, offset by the participant's time zone, on each of the survey's active days.
Reminder Notification Time	The time for a reminder should match the time of its originating notification plus (*index * reminder interval*) minutes, where index is the one-based reminder number, on each of the survey's active days.
Random Notification Gap	The gap between every random notification for the same prompt on the same active day should be at least c minutes, where c is the compliance duration of the prompt.

8.1.1 Computing Active Days

To compute the active days for a survey, attributes on both the survey itself as well as the project need to be examined. The following are the cases for computing survey active days, in order of increasing strength. Bursts are ranges of days that researchers select for certain projects or surveys to be available to participants. The active days for a project are the union of active days for each survey in the project.

Table 9: Cases for computing survey active days

Case	Result
Base	The survey is active on every day for the duration of the project.
The project has bursts	The survey is active on every day of the project's bursts.
The survey has overriding bursts	The survey is active on each day of the survey's bursts.
The survey has no prompts	There are no active days for the survey.

8.2 Algorithm

For schedule creation to be migrated to a cloud-hosted microservice, the scheduling algorithm had to be rewritten from Swift/Java to Typescript. During this process, the algorithm was evaluated and redesigned to work more consistently. The original scheduling algorithm, although mostly consistent, had a few issues. First, the gap between random notifications did not take the compliance duration into account – notifications were always

separated by 15-minute gaps. Although this did not cause issues generally, it made it possible for two different notifications to have overlapping compliance durations. This is an issue because during the overlap it is ambiguous as to which notification participants are responding to, which could lead to a missed prompt or participants being able to respond to the same survey twice back-to-back. In the following example, two notifications with 60-minute compliance durations are only separated by 40 minutes, and therefore have an overlapping compliance period (shown in red).

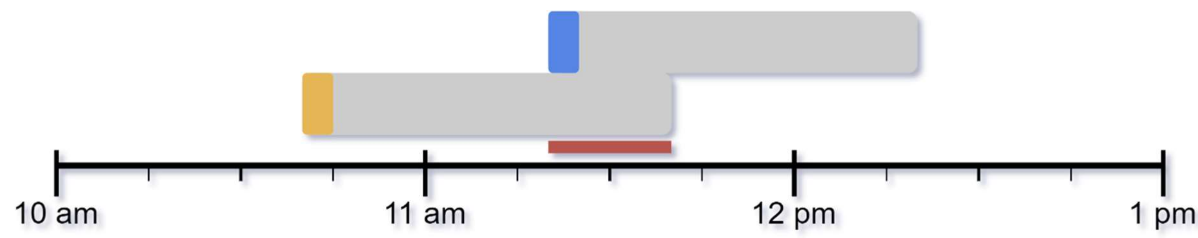

Figure 12: Notifications with overlapping compliance period

The second issue with the old scheduling algorithm was schedule consistency. The algorithm used a greedy approach to scheduling to improve performance. Although this approach was computationally efficient, it allowed for the possibility of creating invalid schedules. The algorithm worked by selecting a random time within the prompt window and scheduling the first notification for that time. It would then 'block-out' 15 minutes before and after the notification and repeat for each notification in the prompt. However, this approach could fail if the total prompt window was less than *15 * number of notifications* minutes long. In the following example, two notifications fail to be scheduled within a 20-minute prompt window, though a valid schedule is possible.

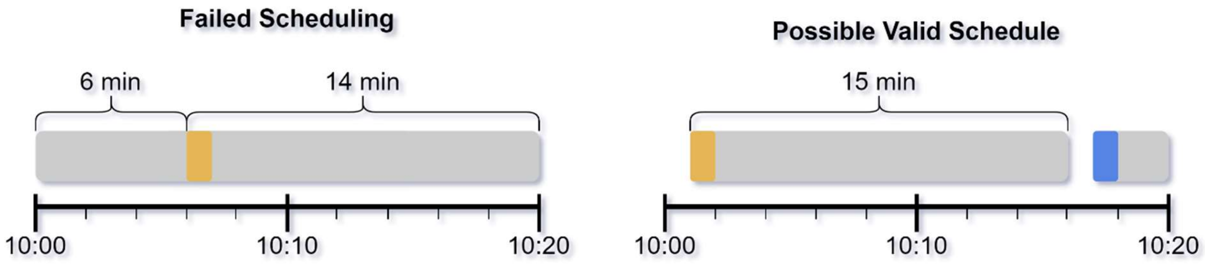

Figure 13: Example of failed notification scheduling

The new scheduling algorithm not only takes prompt compliance duration into account, but also guarantees a valid schedule if one exists. The new scheduling algorithm works recursively to schedule notification. *Note: any tiebreaking is assumed to be random unless stated otherwise*.

1. To start, select the middle notification
2. Temporarily schedule all preceding and succeeding notifications at the earliest and latest possible times, respectively
   a. Each notification should be separated by a gap of c minutes, where c is the prompt's compliance duration
3. Randomly schedule current notification in remaining time slots. This assignment is permanent
4. If the notification includes reminders, schedule them at i-minute intervals after the notification, where i is the reminder interval
5. Recursively schedule the remaining notifications in the two disjoint windows before and after the current notification, excluding the $c - 1$ minutes before and after the notification

The following diagram steps through a simple example of scheduling three notifications with 30-minute compliance durations in a 3-hour window.

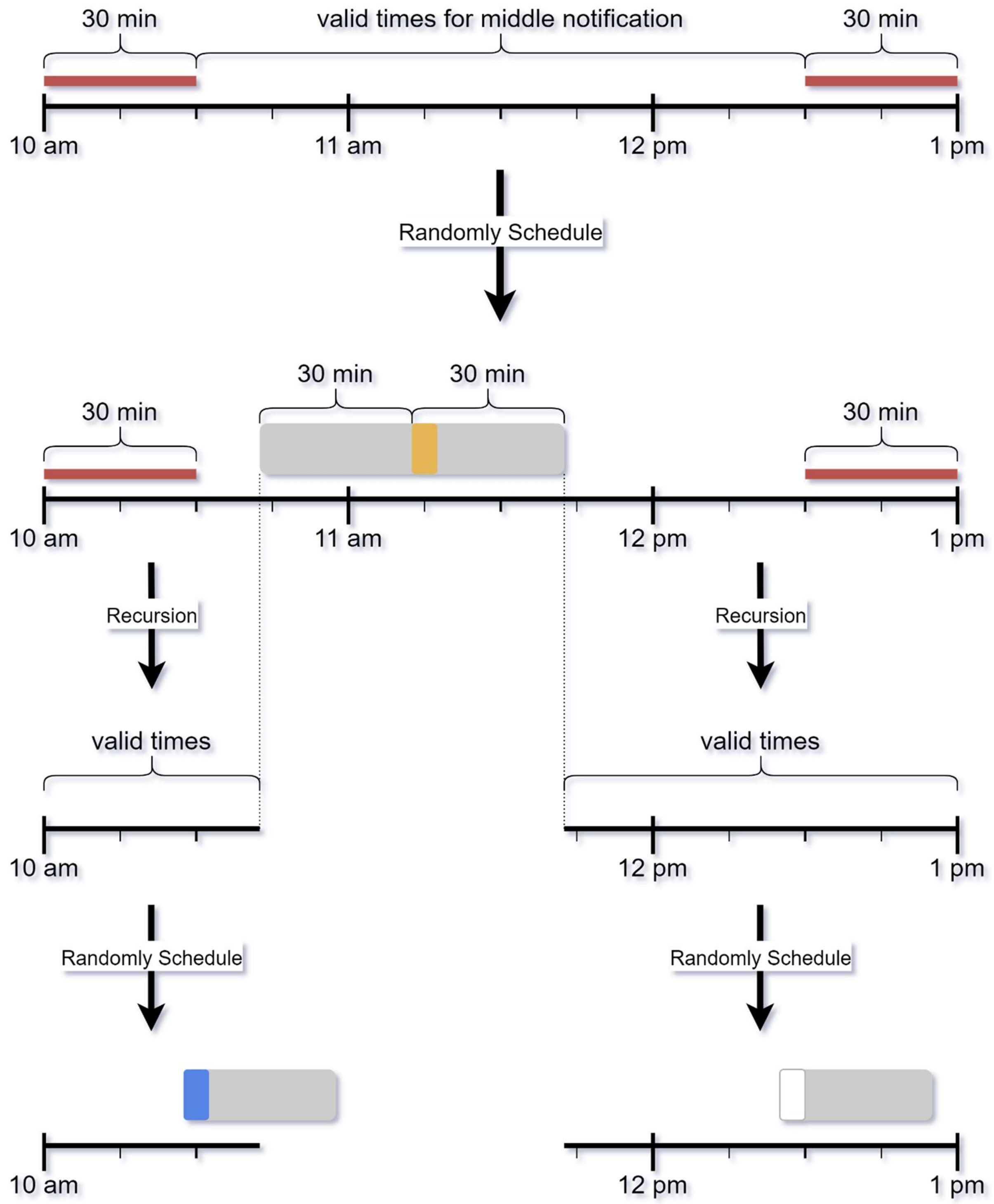

Figure 14: Improved random notification scheduling algorithm

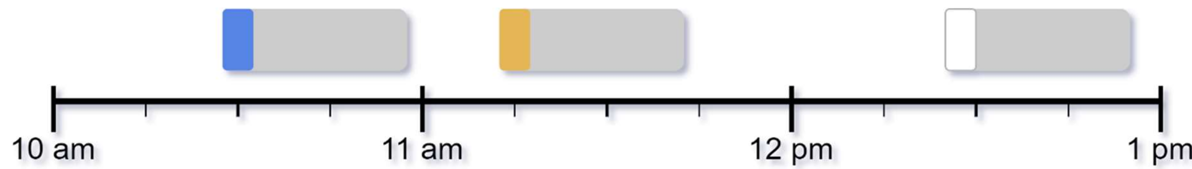

Figure 15: Schedule resulting from the example in Figure 14

The scheduling algorithm presented above not only improves on the consistency of the old algorithm, but also allows TigerAware to derive restrictions that should be placed on prompts during survey creation to guarantee a valid schedule is possible.

Table 10: Derived notification restrictions to guarantee schedule validity

Value	Restriction
Random prompt window	The time window for a random prompt must be at least $c * (n - 1) + 1$ minutes long, where n is the number of notifications in the prompt, and c is the compliance duration of each notification.
Total Reminder Duration	The total reminder duration for a notification must be at most one minute less than the compliance duration. In other words, $n * i \leq c - 1$, where n is the number of reminders, i is the interval between reminders, and c is the compliance duration of each notification.

9. Conclusion and Future Work

TigerAware is a revolutionary application that enables researchers across many disciplines to collect data from their participants in real-time and is especially useful in EMA-style studies. Although TigerAware included all the functionality needed to meet the needs of most users, the implementations included some outdated technologies, inefficient techniques, and complicated workflows.

The work in this project improves the TigerAware platform through upgrades to the existing web dashboard, implementation of a modern microservice backend, and implementation of remote messaging capabilities. These features modernize TigerAware and help to improve the experience of researchers and participants alike. They also improve many existing DevOps practices of the TigerAware team, and enhance development speed, flexibility, and consistency. Overall, these improvements will help TigerAware continue to provide excellent momentary assessment capabilities to researchers for years to come.

9.1 Future Work

As TigerAware moves into the future, there are always more exciting features and topics to work on. Two projects currently in progress in the lab are question-based survey creation and action-based intervention. Question-based survey creation, currently being spearheaded by Charlie Hotrabhavananda, is a new interface to improve the process of creating surveys for researchers. Building a survey on TigerAware's current builder can be a daunting task for researchers new to the platform. There are a plethora of different options and configurations, and it can be difficult to tell which parts are needed for certain protocol

designs. By prefacing this experience with an easy to understand sequence of questions about the survey – *Will your survey have notifications? Should your survey be available all the time? What time will participants receive prompts to take your survey?* – a template can be created to give new researchers a jump-start on their protocol and remove unneeded complexity.

A second feature currently in development is TigerAware's action-based intervention system. Currently led by Logan Harrison, this system will help to further extend researchers' options for participant interaction. The action-based intervention system will allow researchers to configure a set of actions on their surveys which will take place when certain requirements are met. These actions can include a multitude of different integrations from sending researchers an email to updating participants with a push notification to classifying sentiment in an audio recording. This feature can be easily extended to a massive range of use cases, and the sky is the limit for how researchers will be able to use the tool.

Another opportunity for extending the work in this report is to examine further enhancements to the testing capabilities of TigerAware's microservices. The first potential improvement is work on a fully-integrated Continuous Integration (CI) pipeline for both the dashboard and microservices. With the help of the features provided by the Firebase Testing SDK, it is possible to set up a CI pipeline which automatically tests each new commit to the TigerAware repository. This feature could help to further improve system consistency, avoid bugs in production, and decrease developer time required for manual testing and code review.

10. Works Cited

[1] Pew Research Center, "Mobile Fact Sheet," 12 June 2019. [Online]. Available: www.pewresearch.org/internet/fact-sheet/mobile. [Accessed 25 March 2020].

[2] S. Shiffman, A. A. Stone and M. R. Hufford, "Ecological Momentary Assessment," *Annual Review of Clinical Psychology,* pp. 1-32, 2008.

[3] National Institute of Health, "Federal RePORTER," Star Metrics, 2019.

[4] A. A. Stone and S. Shiffman, "Ecological momentary assessment (EMA) in behavioral medicine," *Annals of Behavioral Medicine,* vol. 16, no. 3, pp. 199-202, 1994.

[5] T. J. Trull and U. W. Ebner-Priemer, "Using experience sampling methods/ecological momentary assessment (ESM/EMA) in clinical assessment and clinical research: introduction to the special section.," *Psychological Assessment,* vol. 21, pp. 457-462, 2009.

[6] S. Ravi, "Development of a Wireless Body Area Sensing System for Alcohol Craving Study," University of Missouri, Department of Computer Science, 2013.

[7] R. Shi, "An Enhanced Mobile Ambulatory Assessment System for Alcohol Craving Studies," University of Missouri, Department of Computer Science, 2015.

[8] D. P. Srivatsav, "MTD: Mood Toolkit Dashboard," University of Missouri, Department of Computer Science, 2017.

[9] J. Kanugo, "TigerAware Dashboard: An Improved Survey Generation and Response Visualization Dashboard," University of Missouri, Department of Computer Science, 2018.

[10] W. Xia, "TigerAware Android: An Improved Survey and Notification System," University of Missouri, Computer Science, 2019.

[11] W. Morrison, L. Guerdan, J. Kanugo, T. Trull and Y. Shang, "TigerAware: An Innovative Mobile Survey and Sensor Data Collection and Analytics System," in *Third International Conference on Data Science in Cyberspace*, Guangzhou, China, 2018.

[12] C. Richardson, "Pattern: Microservice Architecture," 2020. [Online]. Available: https://microservices.io/patterns/microservices.html. [Accessed 7 April 2020].

[13] Red Hat, "What are microservices?," 2020. [Online]. Available: https://www.redhat.com/en/topics/microservices/what-are-microservices. [Accessed 7 April 2020].

[14] ResearchStack, "ResearchStack: An SDK for building research study apps on Android," [Online]. Available: http://researchstack.org/. [Accessed 6 April 2019].

[15] ResearchKit, "ResearchKit Framework Programming Guide," 2018. [Online]. Available: http://researchkit.org/docs/docs/Overview/GuideOverview.html. [Accessed 6 April 2020].

[16] Firebase, "Firebase Realtime Database," 28 January 2020. [Online]. Available: https://firebase.google.com/docs/database. [Accessed 7 April 2020].

[17] Angular University, "AngularJs vs Angular - An In-Depth Comparison," 9 June 2017. [Online]. Available: https://blog.angular-university.io/angularjs-vs-angular-an-in-depth-comparison/. [Accessed 7 April 2020].

[18] Angular, "Observables Overview," 2020. [Online]. Available: https://angular.io/guide/observables. [Accessed 8 April 2020].

[19] Firebase, "Test locally then deploy to your site," 7 April 2020. [Online]. Available: https://firebase.google.com/docs/hosting/deploying. [Accessed 8 April 2020].

[20] Heroku, "Usage and Billing," 31 May 2019. [Online]. Available: https://devcenter.heroku.com/articles/usage-and-billing. [Accessed 8 April 2020].

[21] Firebase, "Pricing plans: Start for free, then pay as you go," 2020. [Online]. Available: https://firebase.google.com/pricing. [Accessed 8 April 2020].

[22] Firebase, "Cloud Functions for Firebase," 3 December 2019. [Online]. Available: https://firebase.google.com/docs/functions. [Accessed 7 April 2020].

[23] Apple, "Local and Remote Notifications Overview," 2018. [Online]. Available: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG. [Accessed 12 April 2020].

[24] Apple, "UILocalNotification," 2020. [Online]. Available: https://developer.apple.com/documentation/uikit/uilocalnotification. [Accessed 12 April 2020].

[25] Firebase, "Life of a message from FCM to the device," 26 February 2019. [Online]. Available: https://firebase.googleblog.com/2019/02/life-of-a-message.html. [Accessed 12 April 2020].

[26] Firebase, "Firebase Cloud Messaging," 10 April 2020. [Online]. Available: https://firebase.google.com/docs/cloud-messaging. [Accessed 12 April 2020].

[27] Firebase, "FCM Architectural Overview," 25 February 2020. [Online]. Available: https://firebase.google.com/docs/cloud-messaging/fcm-architecture. [Accessed 12 April 2020].

[28] Firebase, "Unit testing of Cloud Functions," 8 April 2020. [Online]. Available: https://firebase.google.com/docs/functions/unit-testing. [Accessed 16 April 2020].

[29] OpenJS Foundation, "Mocha: simple, flexible, fun," 5 April 2020. [Online]. Available: https://mochajs.org/. [Accessed 16 April 2020].

[30] Amazon Web Services, "Amazon EC2 Auto Scaling," 2020. [Online]. Available: https://aws.amazon.com/ec2/autoscaling. [Accessed 7 April 2020].