TigerAware Conditional Action Engine

An Extensible Framework for Providing Real-Time Intervention to EMA Surveys

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

Logan Harrison

Dr. Yi Shang, Advisor

May 2020

Table of Contents

Table of Figures

Acknowledgements

I would first like to thank Dr. Yi Shang for his support in my research, undergraduate studies, and graduate studies. I would also like to thank Dr. Tim Trull, Dr. Tom Piasecki, and Dr. Denis McCarthy from the Department of Psychology who have provided ample feedback and guidance as I developed this project. I am also grateful for the support from all the professors I have had during my studies.

I want to extend thanks to members of the Distributed and Intelligent Computing Lab for their assistance and camaraderie throughout my tenure in the lab. I would specifically like to thank Connor Rowland for his help and knowledge about the TigerAware cloud infrastructure, Luke Guerdan for his availability and readiness to help whenever there was an obstacle, and Zachary Kipping for his willingness to share his expertise about Angular development. And to everyone else, thank you for making these years in the lab memorable.

I also want to thank Dr. Dong Xu, Dr. Chi-Ren Shyu, and Dr. Yi Shang for taking the time to be on my committee. I am honored to have the opportunity to present to such prestigious and decorated faculty members. I also want to thank them for their direct influence on my studies while at Mizzou and their administrative work to always improve and push the department forward.

Finally, I would like to thank my family and friends who have supported my academic career. I would like to specifically thank my mother for always providing love and support in my studies, even when times were tough. Without the support of these people, I would not be in the position I am today, and for that I am forever grateful.

Abstract

TigerAware is a cross-platform system for collecting information from participants via smartphones in real life over time. The system is inspired by applications in Ecological Momentary Assessment (EMA), but it can be used in research studies spanning many domains. TigerAware is one of the first mobile survey platforms to provide an experience tailored specifically to EMA studies through its extensive features that allow surveys to be prompted randomly, connection with external devices, such as blood alcohol sensors, and extensive participant compliance monitoring. In this project, I propose and implement a new component, called Conditional Action Engine, to provide closed-loop feedback for researchers to enable them to interact with their participants as data is collected. The Conditional Action Engine is a highly extensible framework for triggering configurable actions in real-time based upon participants’ responses to surveys. This will allow researchers to be notified or perform essentially any action within seconds of the response being recorded. A grammar is created to define the bounds in which a researcher may create conditional actions via the web dashboard. This grammar allows the platform to be extensible while also staying robust enough to handle the various types of data TigerAware can collect. Firebase Cloud Functions are also leveraged to provide this feedback in real-time without a costly polling approach. Cloud Functions also give the ability to connect to any third-party service via a RESTful API, thus providing nearly endless possibilities for the Conditional Action Engine.

1 Introduction

TigerAware is a mobile survey platform founded and developed by students and researchers at the University of Missouri – Columbia. TigerAware allows researchers to seamlessly create surveys via a web application dashboard. These surveys can then be deployed to their participants' iOS or Android devices where they can be taken immediately. Mobile survey platforms, like TigerAware, are ideal for ecological momentary assessments (EMA). In fact, TigerAware was explicitly developed for EMA studies after researchers from the Department of Psychology expressed frustration with finding a survey solution that provided all the functionality they needed out of the box [1]. This led to the researchers paying for a custom solution or settling with an imperfect out of the box solution. TigerAware aims to be a pre-built solution that provides the same functionality as expensive custom-built solutions.

EMA studies differ greatly from traditional studies. In a 2006 paper, Moskowitz and Young cite several advantages of EMA studies including the ability for a participant to "report on symptoms, affect, behavior and cognitions close in time to experience, and these reports are obtained many times over the course of a study" [2]. One important aspect of an EMA study is the need to collect data from participants in real time, over time. This is incredibly hard to achieve with traditional survey methods because participants in an EMA study often take multiple surveys in a day with strict timing regulations. Furthermore, it is imperative for the integrity of these studies that the survey protocol is followed diligently. With traditional methods, it is hard to regulate participant compliance and follow a strict survey schedule.

For example, several researchers we are working with have complex survey protocols that would not be possible with traditional methods. Some protocols require a survey in the morning at a scheduled time, a survey at night at a scheduled time, and a randomly scheduled survey at some time in the afternoon. This type of protocol is not feasible with a survey platform intended to be used on a computer, and the control is not available in out of the box mobile survey software, so it must be custom designed. All this functionality is included by default in TigerAware. In addition, TigerAware supports integration with external devices via Bluetooth. One such integration is with the BACtrack BAC monitor, which is used by researchers studying alcohol consumption. Overall, TigerAware meets many needs of researchers whose research requires complex EMA protocols.

2 Motivation

TigerAware provides a revolutionary set of tools for conducting EMA studies easily and reliably. As mentioned, EMA studies focus on collecting the same data from participants over time. These studies can last anywhere from several days to several months and generate a great amount of temporal data about a participant's behavior and tendencies. However, I noticed a disconnect. EMA studies are designed to detect changes in participant behavior in real time, over time. Researchers were using our platform to administer cutting-edge EMA protocols but were still performing data analysis as if it had been a traditional study. After the entire protocol was over, researchers would download all the data and view it (often for the first time). There was no opportunity for researchers to delve into the data and easily detect the type of behavior the studies were intended to identify. I wanted to make researchers an active part of the data collection process – a process which, historically, has been incredibly passive.

We will use an example to look at how researchers can be a more active part of data collection to better illustrate the vision for the Conditional Action Engine. Imagine I am a researcher studying the causes and effects of alcohol abuse. In particular, I am interested in studying what I consider "extreme intoxication," classified as a BAC of above 0.2% (0.08% is the legal limit to drive in most states, so approximately 2.5x that level of impairment). My protocol has three surveys: a survey that randomly gets delivered at some point in the afternoon that collects various information about the participant's mood, a survey that gets delivered every night at 10:00pm, and a survey that gets delivered each morning at 10:00am that asks about the alcohol use the previous night. The nighttime survey asks general information about the participant's alcohol use that night and concludes by utilizing TigerAware's BACtrack sensor integration to record the participant's current BAC.

As a researcher, I want to know when someone's BAC goes above 0.2%, so I can go look at their responses from that afternoon to see if there is anything that indicates they might binge drink. In addition, if someone's BAC goes above 0.3%, I want to personally reach out to them and manually check on their well-being. Currently, to do this I must download the entirety of the survey responses, find the most recent day, and search through each participant's response. This is an incredibly tedious process, even with the aid of spreadsheet software. Doing this daily is impractical for researchers, who are often very busy, and very menial for lab assistants.

The Conditional Action Engine seeks to solve this problem and others that are similar in nature. As a researcher who is leveraging the conditional action, I could configure two conditional actions to handle the above scenario. For the first case, detecting BAC above 0.2%, I can configure a conditional action to retrieve the value from the survey question where the participant interacts with the BACtrack sensor, compare it to my threshold of 0.2%, and if it is higher, I will send myself and all of my lab assistants an email to alert us. For the second case, the same configuration could be used, but with a threshold of 0.3%, and I will send only myself a SMS that alerts me. Once deployed, as soon as a participant submits a response, the conditional actions will be automatically evaluated, then perform the action if the condition is met. This system involves the researcher in the data collection process by alerting them in real time to responses that fit criteria they have specified.

The goal of the Conditional Action Engine is to provide an extensible framework that is highly configurable for researchers. In addition, it should be flexible enough to handle the wide variety of data that TigerAware can collect while being robust enough that the behavior is consistent and controlled. In this project, I lay the groundwork for the Conditional Action Engine and create the basic workflows called the *core actions*. The major components included in this project are the system design, UI additions to the existing TigerAware web dashboard, and the backend architecture of the Conditional Action Engine.

3 System Design

System design is one of the most import aspects in any project. This is especially true for a project that is designed to be used, extended, and iterated on in the future. To build a strong foundation for this project, I spent many hours scrutinizing, with the help of my colleagues, every design I came up with. The design presented in this project is the amalgamation of many previous designs with an explicit focus on extensibility and backwards compatibility with the existing TigerAware system.

3.1 Related Work

Related works in computer science are typically used as starting points or inspiration for new projects. However, in this project I used similar works slightly differently. The inspiration for this project came from the desire of researchers to have more control and information in their studies. Once this desire was clear, early plans for the Conditional Action Engine were discussed from scratch simply as a new feature that would be added to the existing TigerAware platform and was not based on any existing work. Rather, the design was verified by analyzing similar systems in different applications.

The application that had the system most like the Conditional Action Engine was ZenDesk. ZenDesk is an all-in-one customer service and customer relationship management (CRM) solution [3]. One product of interest is their customer service ticketing system. This product aims to automate customer contact through an easy to use web portal. Customers can open a ticket with details of their concern or inquiry, and the company can manage these tickets and assign employees to handle them.

One feature of the ticketing solution is the ability to conditionally automate certain actions. For example, if a ticket has the “open” status and has not been assigned to anyone for at least 3 days, it can be randomly assigned to a team member and alert the appropriate manager via email. As you can see, there are many parallels between this system and the Conditional Action Engine. Furthermore, on one of the documentation pages for their system, Jessica Marasco of ZenDesk says, “The condition statements you create for automations contain conditions, operators, and values” [4]. This is very similar to the conditional actions used in the Conditional Action Engine. In TigerAware, a conditional action contains three major user-facing components: a target value, an operator, and an action. In the ZenDesk system, “conditions” (status is open) are analogous to “target values” (0.3% BAC) in the Conditional Action engine – making the systems nearly identical.

Discovering this application *after* the Conditional Action Engine had been designed provided great validation to the design. ZenDesk is large enterprise software company with a current market valuation of nearly $9 billion (May 2020). It is safe to assume that the system they designed for automating the ticketing process has been successful, or at least operational, given their large user base. Furthermore, it provides validation for the motivation of this project. ZenDesk and TigerAware are both applications that revolve around the collection and organization of data. Before the automated ticketing system, presumably, employees were responsible for manually monitoring their data (customer service tickets in this case). In TigerAware, researchers are also responsible for manually monitoring their data (participant survey responses). The fact that ZenDesk added automation to their system proves that users want the ability for data to automatically perform what were once manual workflows. The Conditional Action Engine seeks to implement this same data-driven paradigm that has been successful for ZenDesk into TigerAware.

3.2 Grammar

As mentioned, conditional actions in TigerAware are made up of three user-facing components: a target value, an operator, and an action. In addition, there is a fourth component, called an aggregator, that is abstracted away from the user. The target value is the value to which the participant's response will be compared. The operator defines how the two values will be compared. The action defines what will happen is the condition is satisfied. The aggregator defines *how* we will extract a value from participant's survey response. In the previous examples, the target value is a *BAC of 0.2%*, the operator is *greater than*, and the action is *send email*. These components are discussed in more detail in section 5.2.

To a human, putting all these pieces together is very straight-forward, but how does one make a machine understand? We must first formalize what a conditional action *is*. This is especially challenging because conditional actions are inherently configurable and flexible.

Conditional actions will be used by many different question types in TigerAware surveys, and these question types deal with different types of data that must be evaluated in different ways. The best way to capture this need to be flexible is to create a grammar, or set of rules, that defines the bounds in which conditional actions may be configured. Figure 1 below defines the syntax used by a valid conditional action. It is important to note that this grammar will change as new extensions are made to the Conditional Action Engine. For example, there are currently only three acceptable values for *action*. Once new actions are added to the platform, the grammar must be updated to include that new action. This represents changes that need to be made to the constants configurations in the web dashboard codebase.

```
ConditionalAction:
        action → 'sendSMS' | 'sendEmail' | 'sendNotification'
        aggregationType → '1Q'
        message → String
        operator → 'eq' | 'neq' | 'con' | 'ncon' | 'gt' | 'gte' | 'lt' | 'lte'
        questionId → String
        value → String | Number
        recipients → [String]
```

Figure 1: Grammar for a conditional action

3.3 Supported Question Configurations

The grammar that was created defines the syntax in which conditional actions are created. However, we must go one step further. We must also ensure that the semantic meaning of each conditional action is sound and bridge the gap between the grammar and its use in the UI. As we have seen, conditional actions can be used in many ways through different configurations. TigerAware also supports many different question types that collect different types of data.

Therefore, not all configurations for conditional actions are valid for all question types. For example, it does not make sense for a question that accepts textual input to be evaluated using the *less than* operator. In addition, each question type has an associated data type: string or number. This must be taken into consideration while configuring the UI because we need to guarantee researchers are only able to configure conditional actions that make sense semantically. To provide context for the following question configurations, the screenshots of the question types from the view of the participant are provided as figures 18-23 in the Appendix.

The following tables describe how the above question types are permitted to interact with the Conditional Action Engine. These properties are all inherent to the question types, and for that reason, they will not change. Because of this, they are defined as constants in the web dashboard codebase. These configuration constants are then used by the UI components to properly regulate how conditional actions can be configured by researchers. For each question type there is an associated data type. In addition, there are operators that can or cannot be used by that question type.

Question Type	**Data Type**
Yes/No	string
Text Field	string
Scale	number
Continuous Scale	number
Number	number
BAC	number

Figure 2: Supported data types for TigerAware question types

Question Type	Equals	> or <	≥ or ≤	Contains
Yes/No	✓	✗	✗	✗
Text Field	✓	✗	✗	✓
Scale	✓	✓	✓	✗
Continuous Scale	✓	✓	✓	✗
Number	✓	✓	✓	✗
BAC	✗	✓	✓	✗

Figure 3: Valid operators for TigerAware question types

4 UI Additions

The next major component of this project is the addition of the UI components for the Conditional Action Engine into the existing TigerAware web dashboard. This was by far the most time-consuming part of the project. The UI needs to be intuitive so researchers who are new to using it are not overwhelmed. In addition, it needs to be flexible enough to allow the wide variety of configurations the system supports. The combination of intuitiveness and flexibility complicates the design process a significant amount. In addition, the UI needs to be backwards compatible with the existing system, so current versions of the application do not crash when this new system is added. The sections in this chapter describe the process from basic UI concepts, through design and implementation.

4.1 Considerations

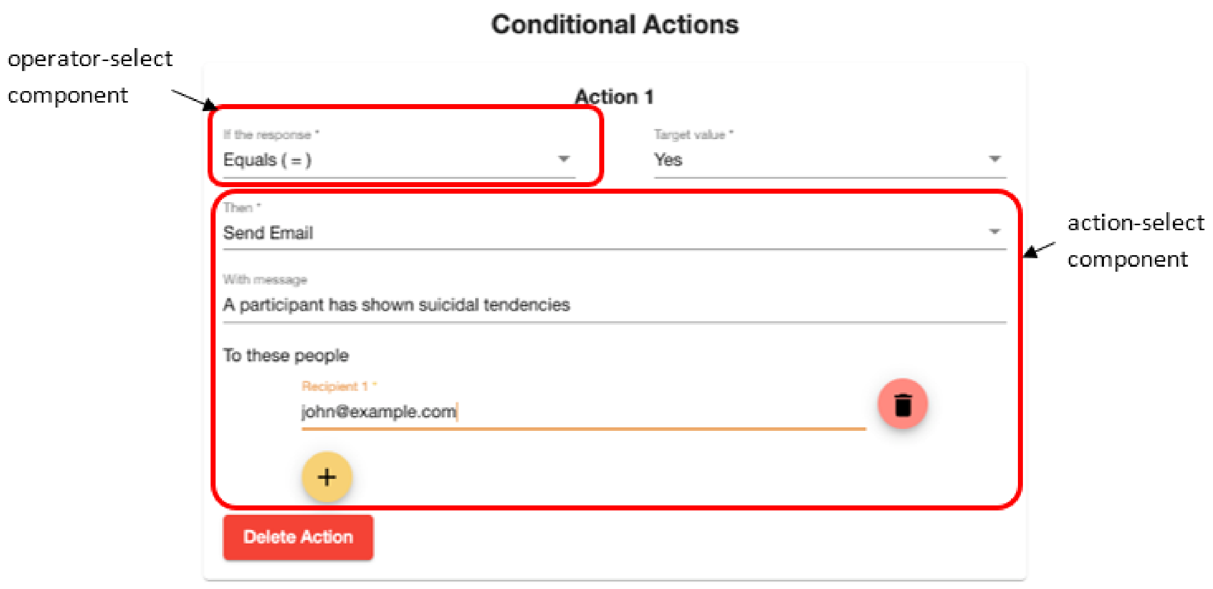
When designing and implementing the UI, there were several themes that needed to be kept in mind. First, TigerAware is primarily maintained by members of the Distributed and Intelligent Computing Lab. Therefore, developers frequently come and go. This presents numerous challenges. One of these challenges is on-boarding new developers to the platform. To keep this process smooth, we need to make sure the new code added to the web application for this project follows the same design patterns as the rest of the application. TigerAware is built using Angular, Firebase, and microservices – technologies that most undergraduate (and graduate) students have very little experience with. Following the same programming patterns as the rest of the application will reduce confusion during on-boarding by only forcing new developers to learn one major design pattern. In addition, the patterns used in the dashboard project were designed by Zachary Kipping, who now travels the country doing workshops for enterprise clients who are switching to a technology stack based on Angular and/or Firebase, so they are a good guideline to follow.

Another challenge that would present itself in this portion of the project was the shear size and complexity of the TigerAware dashboard codebase. The dashboard allows researchers to configure nearly every aspect of their studies which are highly configurable. This leads to a very complex codebase to capture all the different settings and options available to researchers. Furthermore, Firebase is a realtime, NoSQL database [5]. This further complicates the code for the dashboard because JavaScript streams must be used to display data to the user in real time as it is modified in the database. To modify this data as it is streamed to the web application, functional programming techniques must be employed to transform the data to the appropriate form for display on the UI. To do this, a library called "RxJS" (short for reactive JavaScript) is utilized because it was created for this exact purpose [6]. These technologies are discussed in more detail in section 5.1, but they are worth mention here because of the added complexity it brings to the frontend of the web dashboard.

To minimize added complexity to an already complex codebase, the last consideration that was made in designing and implementing the UI was to reuse as much of the existing code as possible. Furthermore, the new code that is added for this project should by highly reusable between the new components. This notion has several advantages in addition to avoiding further complicating the codebase. The first advantage is it is simply less code to maintain. As previously discussed, many different people are working on many different parts of this TigerAware project. By adding less code, it is less of a hassle for future developers to maintain. By keeping the codebase concise, developers in the future will not be "spread thin" trying to maintain a massive codebase. The second big advantage of code reusability is fewer places for bugs to be introduced. TigerAware is available as commercial software for researchers to conduct studies that are funded by various organizations. It is imperative that bugs are not introduced into the system. Deploying code with bugs to researchers conducting these studies could, in the worst case, discredit their results and force them to start data collection again. This is an expensive process in both time and money. Therefore, any opportunity to reduce the introduction of bugs into the system should be taken.

4.2 UI Components

After considering the general principles from the previous section, along with the supported question configurations from section 3.3, three main UI components were designed. These components encompass the current, and future, needs of the Conditional Action Engine. The three main components differ only in the input method for the target value and they are: dropdown, text box, and number box. In addition, the operator-select and action-select portions of the UI are separated into their own components which are reused through all the possible UI configurations. This design of the components minimizes lines of code added to the project while maximizing code reusability. A visualization of how the UI is split into different components is shown in figure 4.



If the response *
Equals (=)

Target value *
Yes

Then *
Send Email

With message
A participant has shown suicidal tendencies

To these people

Recipient 1 *
john@example.com

Figure 4: Visualization of nested operator- and action-select components

As shown previously, not all question types support all operators. Therefore, the operator-select component needs to take many forms. This is accomplished by utilizing the supported question configurations that are stored as constants in the dashboard codebase. For each question type, there is a list of the supported operators. When the operator-select component is constructed, a list of the valid operators is passed as a parameter to the component, so the proper operators can be displayed for the different question types. Figure 5 below shows some of the many forms the operator-select component can take. Each operator has a user-facing label that is displayed in the list and a corresponding string value that matches the operators defined in the grammar.

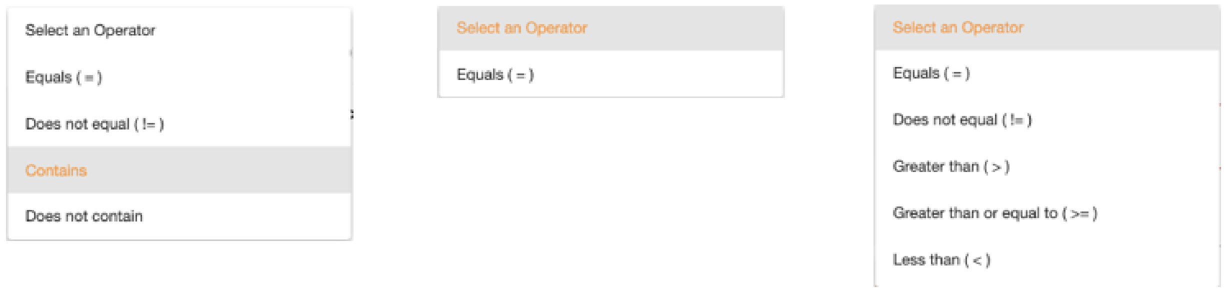


Figure 5: Operator-select component with various valid operators

Unlike the operator-select component, the action-select component behaves the same regardless of which question type for which the conditional actions are being configured. The action-select component contains three major components: the action, the message, and the recipients. The component is designed to read similarly to an English sentence to help make the component more intuitive for new users. For example, in figure 4 the configuration reads "If the response equals yes, then send an email with message 'A participant has shown suicidal tendencies' to john@example.com." The list of actions is stored as a constant in the project and simply accessed by the component upon construction. In figure 6 you can see the dropdown options for the desired action.

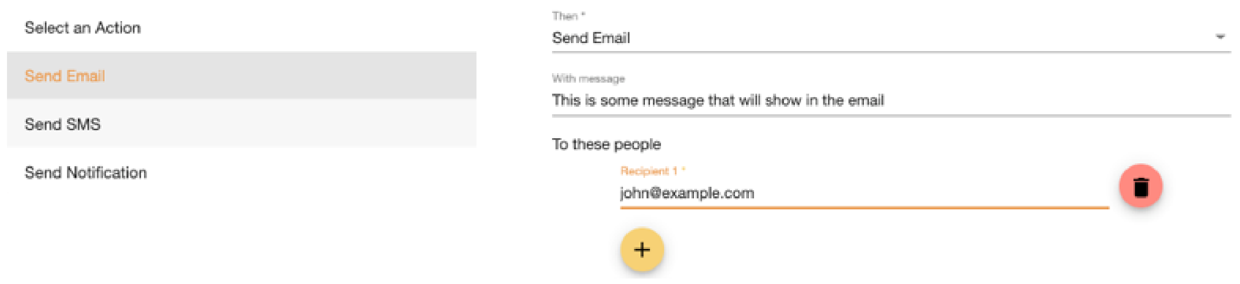
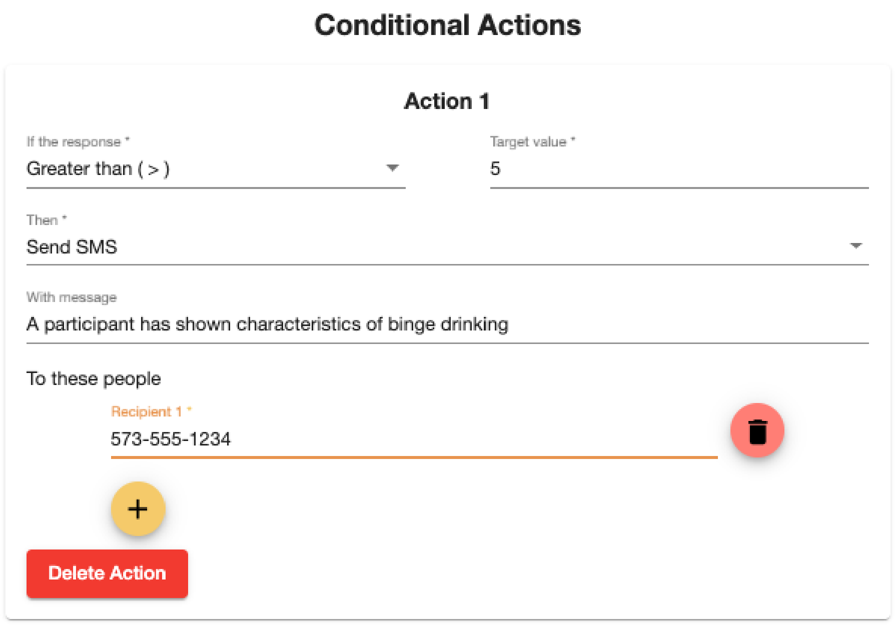


Figure 6: Action-select component with current action options

The operator- and action-select components are utilized by the three main UI components. The first UI component to be discussed is the number box conditional action component. This component utilizes a number type HTML input and can be seen in figure 7. The component is utilized by the continuous scale, number, and BAC question types. One important configuration for this component is the ability to add bounds to the number input. For example, a conditional action for a BAC question type should not accept an input below zero because a BAC value will not be below zero. In addition, the continuous scale question type sets both an upper and lower bound, so these bounds should also apply to the conditional action input. These are optional parameters passed to the constructor for the component. They are optional because for the number question type, there are no inherent bounds that need to be set.


Conditional Actions

Action 1

If the response *: Greater than (>)

Target value *: 5

Then *: Send SMS

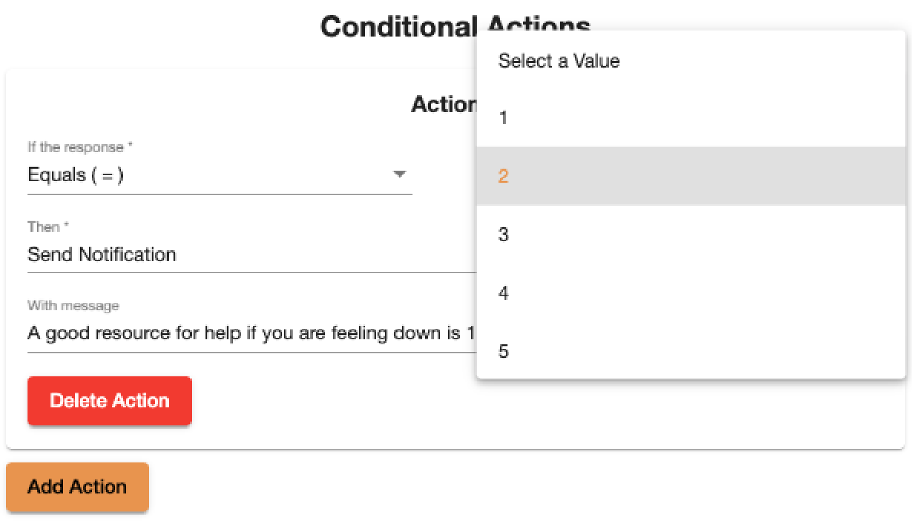
With message: A participant has shown characteristics of binge drinking

To these people

Recipient 1 *: 573-555-1234

Figure 7: Number box conditional action component

The next component is the dropdown conditional action component. This component utilizes a dropdown menu that allows users to select a value from a pre-determined set of values. This component is used for the scale and yes/no question types. When constructed, a list of dropdown options must be passed as a parameter. For the yes/no question, this is simple because it is a static list of either “yes” or “no”. However, for the scale question type the dropdown options must be dynamic based on the bounds set for the question. This requires passing a parameter that is an Observable type so new values are passed through as they are changed in the scale question configuration. An example of this component’s use in a scale question type is shown below in figure 8.


Conditional Actions

Action

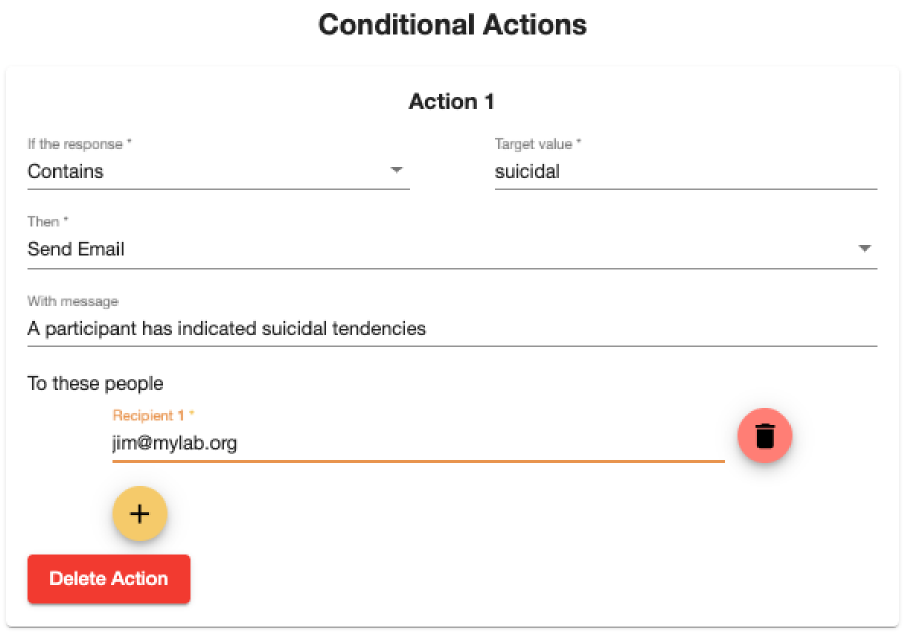
If the response *
Equals (=)

Then *
Send Notification

With message
A good resource for help if you are feeling down is 1

Figure 8: Dropdown conditional action component

The last component is the text box conditional action component. This is the simplest of the three components because the input is a simple HTML text input. In addition, there is no validation needed (other than the existence of a value) because the text value can realistically be anything needed by the researcher. Currently, the component is only used by the text field question type. An example of this component is shown below in figure 9.



Conditional Actions

Action 1

If the response *
Contains

Target value *
suicidal

Then *
Send Email

With message
A participant has indicated suicidal tendencies

To these people

Recipient 1 *
jim@mylab.org

Figure 9: Text box conditional action component

Overall, we have seen that with the addition of five new components, we have covered all configurations currently supported by the system. In a minimal amount of additional code, the UI contains all necessary controls researchers need to configure and use conditional actions in their projects. The ability to reuse the operator- and action-select components drastically cuts the amount of development time that was needed and prevented duplicated code being added to the codebase. Figure 10 below shows a summary of the UI component utilized by each of the six currently supported question types.

Question Type	UI Component
Yes/No	dropdown
Text Field	text box
Scale	dropdown
Continuous Scale	number box

Number	number box
BAC	number box

Figure 10: UI components for TigerAware question types

4.3 Implementation

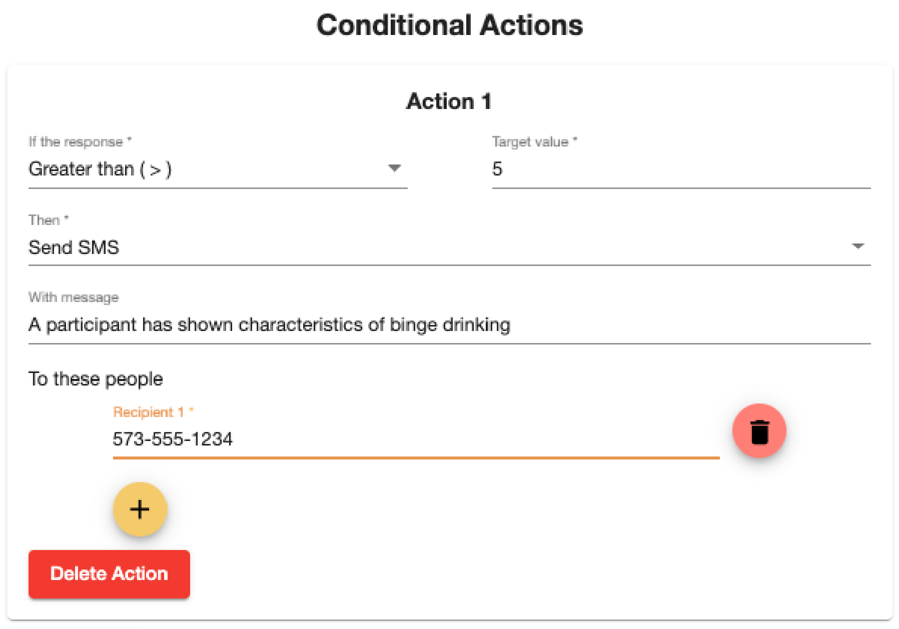
The UI components above were developed by utilizing Angular components, reusable snippets of code that serve one purpose [7]. In addition, the UI is styled using Angular Material. Angular material provides pre-built components that meet material design standards that can easily be used in an Angular project to style and add basic functionalities [8]. For example, each action is embedded in a *material card*. This gives the appearance that the content is layered on top of the background – a key component of the material design standard. Angular material also provides a set of icons which can be used in the project to make basic actions more intuitive. As shown above in figure 9, a trashcan and plus icon are used to represent deleting a recipient and adding a recipient. This allows the design to remain uncluttered while remaining intuitive. The styling for the dropdowns and buttons is also included in Angular Material. Angular Material is used throughout the entire web application, so its use in the Conditional Action Engine UI was paramount for design continuity.

Angular reactive forms are used to represent data throughout the web application. Because everything on TigerAware is configurable by nature, maintaining the data in an Angular reactive form allows us to easily track and validate user input – an essential step to ensure the system behaves as expected [9]. Reactive forms allow developers to specify certain criteria that must be met for certain user inputs. For example, through the use of a pre-built *form validator*, I can ensure that the input is at least 8 characters long [10]. The reactive form object has several metadata properties that provide insight to the current state of the form. If a validator is not satisfied, a property in the form metadata will represent that the entire form is invalid. Forms can be made up of many elements, including additional forms, and for the form to be valid, all children inside of that form must also be valid.

The above fact proves very useful in TigerAware. The entire question configuration is stored as a form with many forms contained within as children. One example that proves the usefulness of this is when a question is being edited. The button to save all changes can be disabled whenever the top-level form is invalid. This means that somewhere inside of that form, the user entered an illegal value. Thus, they cannot save their changes, preventing bad user input. In addition, nested forms are used heavily by the conditional action UI components. The entire component data is stored in a form on the containing question. Within the form for the conditional action components are two fields and a nested form. The fields represent the operator and the target value, while the child form is used to represent information about the action. Within in the action form, validators can be dynamically set on the list of recipients based on if the action is *send email* or *send SMS*. This is very valuable because it validates the recipients as they are entered and alerts the user if they enter an invalid value. Additionally, nesting the form inside of the question form means very little code was changed regarding question validation. Since validation is defined recursively within forms, the new changes were completely backwards compatible due to the validation metadata that is stored on the top-level form. Without Angular reactive forms, validating user input for this project would be much more difficult and prone to bugs.

5 Backend Architecture

The backend for the Conditional Action Engine is the workhorse that powers the incredibly diverse courses of action that can be triggered automatically by the system. The goal of this backend architecture is to utilize microservices and database triggers to provide a robust and extensible way to automatically handle participant responses. An example is shown to provide context into what type of data is stored for conditional actions. This data model is important because it is what is used in the pipeline when the backend needs to evaluate conditional actions. As shown in figure 11 below, a conditional action has been configured by a researcher. In figure 12, you can see the result of that configuration, i.e. what is stored in the database and used at evaluation time. Aside from the *aggregationType* and *questionId* fields, everything else should look familiar and is mapped relatively intuitively. In this chapter, the rationale behind and the use of these additional fields will be discussed. In addition, it will be shown how these fields are essential to the current operation of the pipeline as well as its future extensions.


Conditional Actions

Action 1

If the response *: Greater than (>)

Target value *: 5

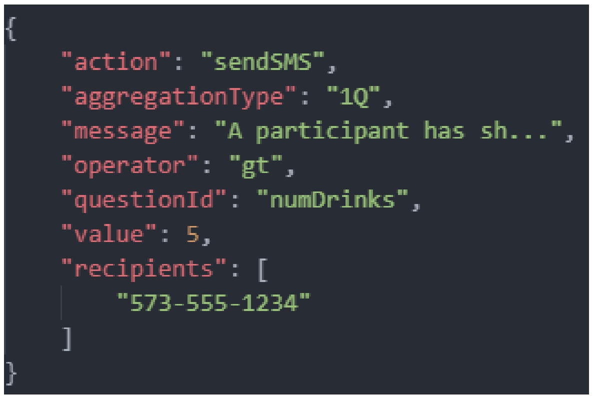
Then *: Send SMS

With message: A participant has shown characteristics of binge drinking

To these people

Recipient 1 *: 573-555-1234

Figure 11: An example conditional action configuration



```
{
    "action": "sendSMS",
    "aggregationType": "1Q",
    "message": "A participant has sh...",
    "operator": "gt",
    "questionId": "numDrinks",
    "value": 5,
    "recipients": [
        "573-555-1234"
    ]
}
```

Figure 12: Example mapping of conditional action using grammar

5.1 Technologies

The natural starting point when discussing any new architecture is to outline the existing architecture and technologies. Figure 13 below shows the current architecture of TigerAware. The main feature of this architecture is the extensive use of Firebase Cloud Functions to handle "server"-side operations – though there is not a server we must maintain since it is hosted in the cloud. Firebase Cloud Functions are "a serverless framework that lets you automatically run backend code in response to events triggered by Firebase features and HTTPS requests" [11]. One of these "events triggered by Firebase" allows us to watch a specific part of the database for specific actions. These are referred to as database triggers. Triggers are used across TigerAware to maintain data integrity since Firebase, a NoSQL database, does not have the data integrity constraints that a typical SQL database would have. In addition, these actions trigger in real time (on the order of milliseconds). The Conditional Action Engine uses triggers to initiate the evaluation process by watching for new participant responses to be written to the database – called an "on create" trigger.

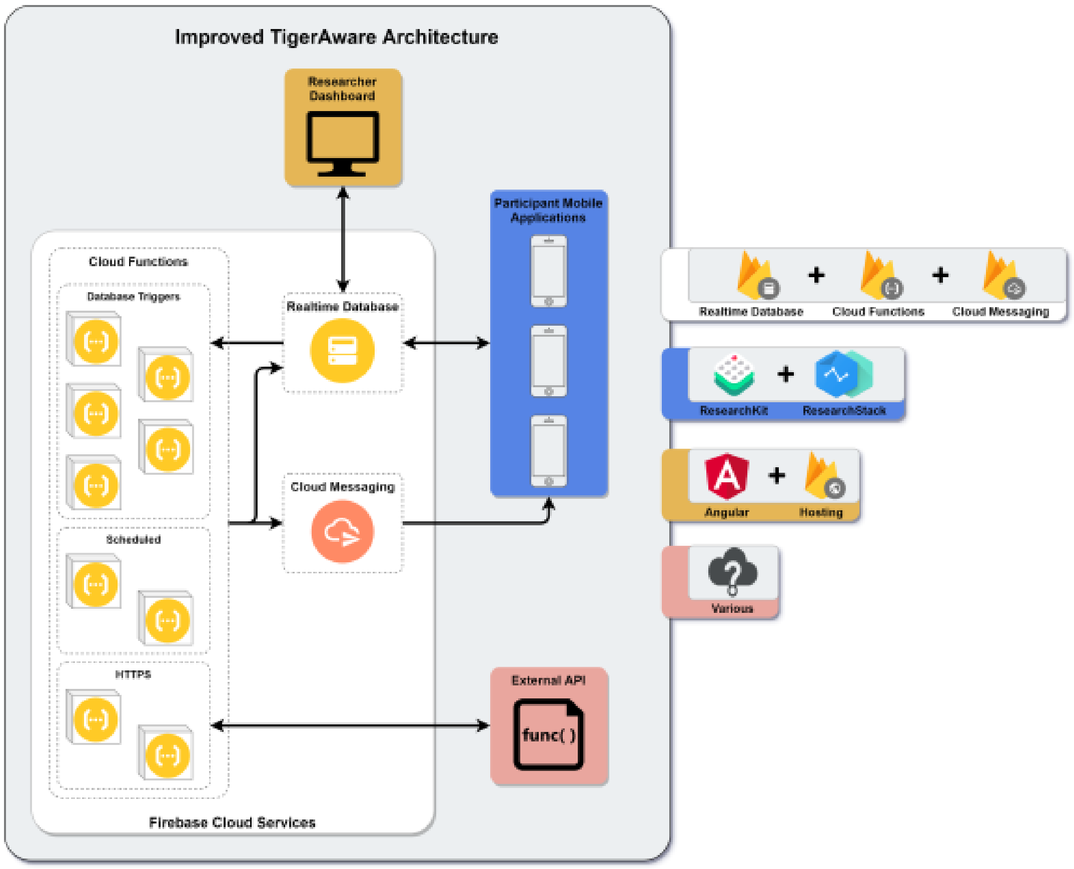


Figure 13: Current TigerAware architecture

The next feature of the architecture of relevance is the microservice pattern. Microservices are also implemented with Cloud Functions, much like database triggers were. A microservice simply refers to a small piece of code that performs one and only one action. This allows for easy debugging as they cleanly separate code out into independent pieces with no side-effects. Microservices also allow for easy code reuse. For example, in the Conditional Action Engine, one possible action is the send a notification to participants. This functionality already exists in TigerAware. Therefore, the Conditional Action Engine can simply call that microservice via HTTP with the appropriate parameters, and the notification will be delivered. This plays an important role in the Conditional Action Engine also when creating new actions like sending email and SMS.

Along those lines, the final architecture feature allows for the possibility of sending email and SMS easily. Cloud Functions can easily access any third-party API via HTTPS. This is paramount for the Conditional Action Engine because it means there are nearly endless possibilities to add as options for actions. In section 5.4, the benefit of being able to utilize REST APIs from Cloud Functions will be very clear while discussing the implementation of the send email and send SMS services. Essentially, cloud functions provide immense flexibility for developers to leverage other technology, while scaling *very* well.

5.2 Conditional Action Pipeline

The biggest breakthrough of the project, and the most critical part, was the development of the conditional action pipeline. This pipeline, shown in figure 14 below, defines the way in which conditional actions are evaluated. There are three main parts to the pipeline: the aggregator, the operator, and the action. The aggregator is responsible for retrieving and formatting data in a way that can be understood by the operator. The operator then evaluates that value against the configured target value using the configured operator. Finally, the action step is responsible for routing to the correct service for performing an action and formatting the appropriate parameters for that service.

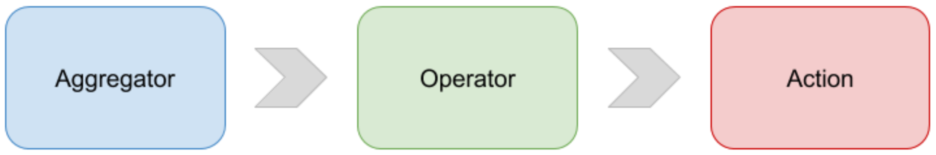


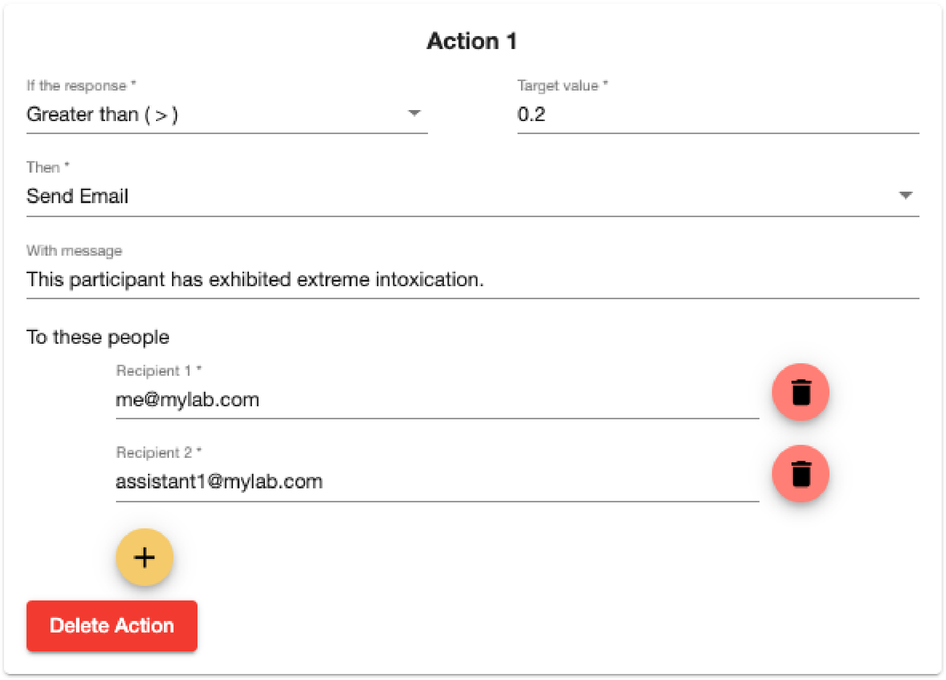
Figure 14: Conditional action pipeline

The aggregator is the solution to perhaps the hardest question that had to be answered when designing this whole system: "How can we provide room to extend the system in a way that the core fundamentals of the platform stay the same from as they are from day one?" This question was incredibly difficult to answer at first because it seems so easy to get the data you need: just go to the database and get it. However, designing the system like this would "handcuff" it in the sense that it would have to be re-developed to tackle more complex use cases. The aggregator design is *intentionally* over-engineered for now, so extensions in the future have a convenient and well-defined interface to plug into. The design should allow developers in the future to simply adhere to the defined API and perform a wide variety of data aggregation techniques.

The operator and action steps have a much more straightforward job than the aggregator. The operator simply compares the output of the aggregator to the target value the researcher specified using the operator the researcher specified as well. While it is straightforward, there are numerous precautions taken in this step. The first precaution is to guarantee the data is the type that was expected. For example, comparing a string to a number should not accidentally happen – that would impact the predictability and reliability of the system. The second precaution during development was testing the operators with the different data types. JavaScript is notorious for having unexpected results with some comparison operators, so diligently testing the code before deploying is essential. The action step simply checks the output of the operator. If the operator indicated that the response met the criteria, the action handler routes to the appropriate service to handle that action. This step is discussed in more detail in section 5.4.

5.3 Workflow

To provide insight into the overall workflow of the Conditional Action Engine, this section will use the BAC example from the Motivation chapter as an example of how conditional actions are evaluated. In figure 15 below, you can see the UI configuration that corresponds to the example. Figure 16 shows the corresponding conditional action data that will be stored in the database for the given survey. This is the data that is available for use while evaluating the conditional actions. Finally, figure 17 depicts the steps that the Conditional Action Engine performs while evaluating the conditional actions.



Action 1

If the response *
Greater than (>)

Target value *
0.2

Then *
Send Email

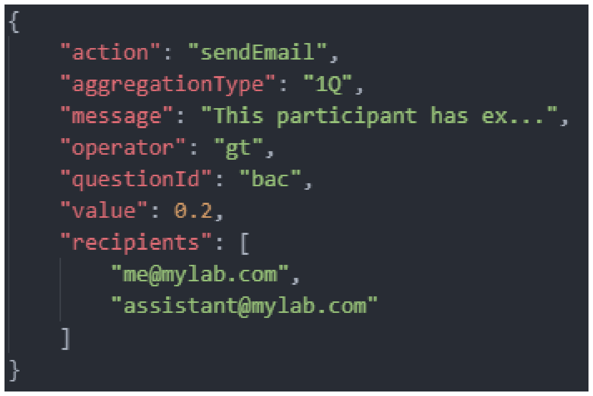
With message
This participant has exhibited extreme intoxication.

To these people

Recipient 1 *
me@mylab.com

Recipient 2 *
assistant1@mylab.com

Figure 15: Example conditional action configuration for workflow demonstration



```
{
    "action": "sendEmail",
    "aggregationType": "1Q",
    "message": "This participant has ex...",
    "operator": "gt",
    "questionId": "bac",
    "value": 0.2,
    "recipients": [
        "me@mylab.com",
        "assistant@mylab.com"
    ]
}
```

Figure 16: Example mapping of conditional action using grammar for workflow demonstration

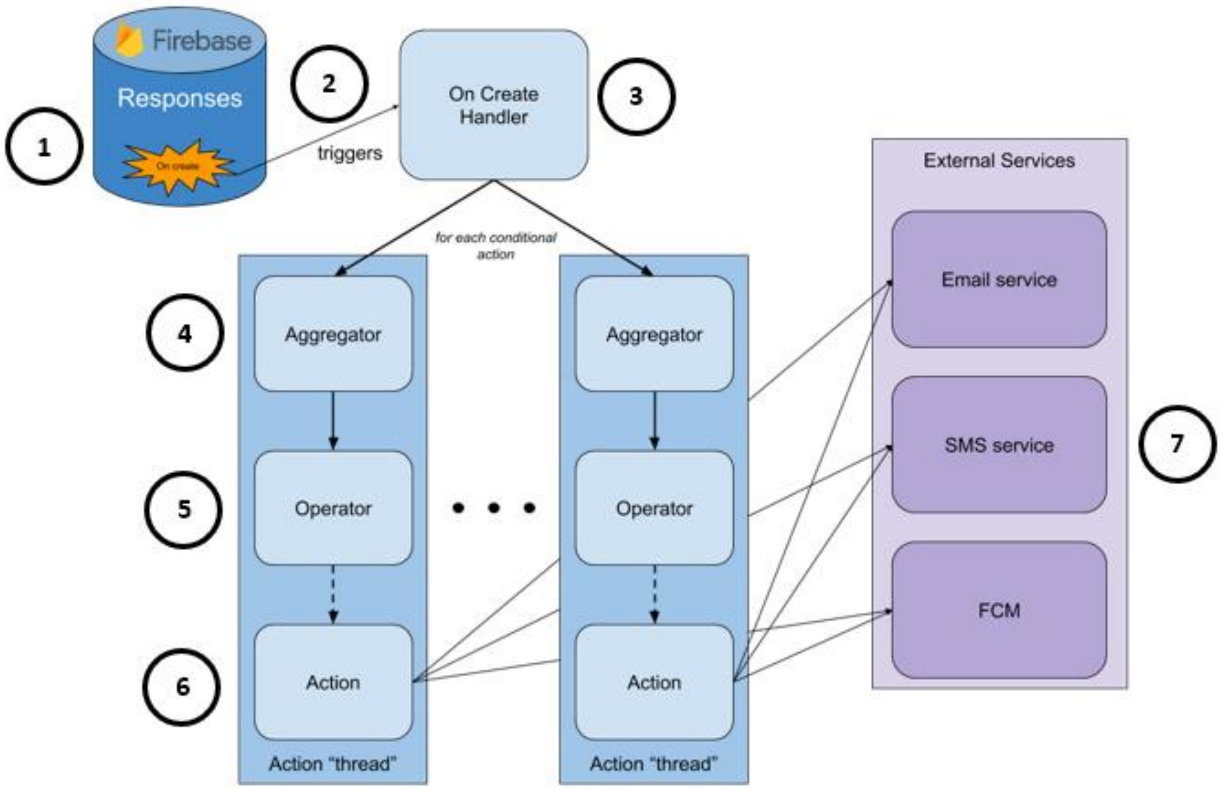


Figure 17: Conditional Action Engine evaluation workflow

1. In this step the user just created a response to a survey in which they are currently enrolled. When a user submits a response, they are stored in a database location which includes the survey key in its path. For this example, say the survey has only one question: the participant uses the Bluetooth BAC sensor to record their BAC. Assume the participant records a BAC of 0.24%. This value is stored in the survey response object with the key of the question id. In this case, the researcher has configured the question id to be "bac".

2. When this response is recorded, the *on create* database trigger fires. Recall that database triggers watch a certain location in the database for certain actions to happen. In the

Conditional Action Engine, the triggers are configured to watch the path ‘`data/{surveyKey}/answers/{responseKey}`’. The *surveyKey* and *responseKey* are wildcards that will match any value. Additionally, those tokens become variables which contain the value they matched. This allows the trigger to have context of the event that just fired. This begins the execution of our cloud function.

3. Next, the list of conditional actions is retrieved. TigerAware has a concept called *blueprints*. A blueprint holistically defines every aspect of a survey including the questions in the survey and other metadata. The conditional actions are stored in this metadata. The JSON representation of the conditional action as shown in figure 16 exists in an array on the blueprint object in Firebase. This blueprint object is keyed on the *surveyKey* from the context of the database trigger. Using this key, we can retrieve the list of conditional actions for the survey. If the list is empty, there is no need to continue so function execution stops. In this case, there is one conditional action in the list, so execution would continue.

4. For each conditional action present, the conditional action pipeline is started. Furthermore, since the pipeline includes many asynchronous operations (database reads, HTTPS calls), each conditional action is handled in parallel. The first step in the pipeline is the aggregator. The aggregation type is retrieved from the conditional action which is then used to determine which aggregator function should be used to aggregate the data. In this example, the aggregation type is “1Q” which stands for single question. This is the simplest aggregator possible and currently the only supported aggregator. The function for this

aggregation type simply retrieves the response value for the provided question. It takes the response key and question id as parameters and returns the value from the response object from the database. In this example, the single question aggregator function would retrieve the value for question “bac” which would be 0.24.

5. Now, the operator takes the output of the aggregator, 0.24, and evaluates it against the target value by way of the operator. Before this is done, however, the data types are checked to make sure they are the expected types. In this case, the operator, greater than, expects the target and response values to be of type number. Both values are of number type, so we continue. The response value, 0.24, and target value, 0.2, are compared, and since 0.24 is greater than 0.2, the operator returns true.

6. The action step first looks at the output of the operator. If the operator returns false, execution stops because the participant’s response did not meet the specified criteria. If the operator returns true, the action step can continue processing the conditional action. The first requirement is to retrieve the action from the conditional action. In this case the action is “sendEmail”. Before routing to the email service to send the email to the recipients, the parameters must be generated. For an email, a subject, body, and recipient list need to be generated and passed to the email service. The subject is simply a constant property defined in the codebase that reads “TigerAware conditional action triggered.” The recipient list is already defined by the researcher, so that can simply be used as-is. Last, the body must be generated. The body includes useful information about the action that was triggered

including the survey name, the participant, the question, the participant's response, and the message that the researcher configures. In this example, the parameters would be generated then passed to the email service.

7. In the last step, the action is performed by the corresponding microservice. Currently, there are three supported actions: send email, send SMS, and send notification. The send notification service already existed in the system, but the send email and send SMS services are new in this project. For the current example, an email would be sent to the two participants via the NodeMailer API that will be discussed in more depth in the following section.

Overall, this workflow provides a clear and efficient way of evaluation conditional actions. By separating each step into a simple input-output problem, a lot of complexity is simplified out and the process becomes very clear. In addition, by utilizing asynchronous JavaScript techniques like async/await and promises all conditional actions for a survey can be performed in parallel. Not only is this beneficial for performance, but Cloud Functions are priced based on runtime [12]. The workflow is also extensible to new aggregators and actions. Since each step in the process performs just one task, if the addition follows the same programming patterns it should fit in seamlessly.

5.4 External Services

A major part of this project was the addition of two new services to TigerAware. The ability to send emails and SMS to users is a widely popular feature found on apps spanning many domains. Rather than simply "hardcoding" the services to work for the Conditional Action Engine, a generic interface was set up to utilize the two new services. Now, different parts of TigerAware can utilize these services to send any message to any person. The services both utilize the ability of Cloud Functions to access third-party APIs via HTTPS. While this section only covers the two services that were added, the following chapter discusses a few extra ideas of other services that could be utilized from Cloud Functions.

The first major service added in this project was the ability to send email programmatically. NodeMailer is a popular JavaScript library that allows users to send email via an HTTPS call [13]. This library is particularly powerful because it allows users to send email without configuring a SMTP server. This is very inconvenient, especially for TigerAware. Recall that TigerAware is hosted entirely in the cloud with no servers managed directly. Creating a SMTP server in the typical fashion would break the clean architecture that currently exists. In addition, this would add cost to running TigerAware. NodeMailer is free to use and bypasses the need to create your own SMTP server – all you need is a working email account from a service like Gmail. NodeMailer uses your email credentials to access the SMTP server of your email provider. Not only is it free and easy to use, but it is also secure. NodeMailer uses TLS to connect to the SMTP server. NodeMailer *does* need to be given access to the email username and password, but Firebase provides a way to store these securely in project-wide environment variables [14]. Furthermore, a separate email account was created specifically for this purpose that has a unique password not reused anywhere else. Finally, NodeMailer provides an easy-to-use library for JavaScript that contains convenient wrappers for the HTTP calls being made.

The second major service added in this project is the ability to send SMS. Unfortunately, there is not an equivalent SMS version of NodeMailer. That is, there is not a free API to send SMS that meets the needs of TigerAware. Instead, the Twilio API was used instead. Twilio provides a similar interface to NodeMailer: simply provide what you want to send, who you want to send it to, and then send it. It was incredibly easy to use [15]. They provide a nice library for JavaScript that has wrappers around the HTTP calls, much like NodeMailer. As mentioned, it is not a free API. The pricing is two-fold: "leasing" the phone number and a per-message cost. At the low end of the spectrum, a local phone number leases at $1 per month and costs $0.0075 per message sent [16]. This plan has a limit of one message sent per second, but this fits well within the current needs of TigerAware. The Twilio API does not support sending the same message to multiple people in one API call like NodeMailer does. To make the interface match that of the NodeMailer interface, I add a wrapper over the library function that accepts a list of phone numbers and makes subsequent calls to the Twilio API. This will allow other TigerAware developers to use the two new services in the exact same fashion, further proving their accessibility for new features.

6 Future Opportunities

As we have seen, the Conditional Action Engine is very robust and flexible – designed with the future in mind. There are nearly endless ways this system could be used and adapted for future studies conducted using TigerAware. The design of the system allows the actions module to be particularly extensible. Any service that provides a REST API accessible via HTTPS can be utilized by the Conditional Action Engine. Some interesting ideas here would be to schedule meetings via a Gmail API or to connect to smart homes via smart speakers like Google Home or Amazon Echo. Furthermore, there are numerous opportunities for more advanced data aggregators. The following sections give a brief overview of useful aggregators that could be added.

6.1 Historical Aggregators

A historical aggregator encompasses any data aggregator that would use previous responses from the participant – not just the current response. These aggregators would be useful because EMA studies focus on detecting changes in participant behavior over time. Two research questions that could be answered in real time by using these aggregators are “Is this the first time a participant has answered a question this way?” and “Does this response seem *normal* for this participant?” Being able to answer these types of questions in real time would be extremely valuable in EMA studies since they provide great insight into the behaviors that are being studied. I will provide a few examples of how these could be used in practice because I believe this is the most intuitive aggregator to the platform.

Say I am a researcher studying mental health disorders and I have a survey question that reads “Are you happy?” I am interested in knowing if someone answers this question in a way they have not answered before (after they have recorded at least three responses). The current aggregators cannot support this. I cannot set an action to trigger on “yes” or “no” because the value I am looking for is dependent on previous values. In this case, I would need a historical aggregator that pulls all previous responses for that question from the participant and checks if the most recent response appears in the list. This aggregator could be called a unique value historical aggregator and would take a minimum number of responses as a parameter. It could simply return a boolean to indicate if the value were unique. Then, an *equals* operator could be used to check if it is true.

The previous example uses discrete values as a response. However, TigerAware supports continuous numerical data as well. To deal with continuous data as a researcher, I want to answer the question “Does this response seem *normal* for this participant?” Say I have a question that reads "How many drinks have you had tonight?" To detect a change, I need to know how likely it is that the participant would answer the way they did. One way to do this is to return the number of standard deviations away the latest response is from the previous responses. This aggregator would take the minimum number of responses as a parameter and return the number of standard deviations away from the average the latest response was. I could then use a greater than or less than operator to compare to a set number of standard deviations that I configure. For example, if the person has answered the question with responses consistently between zero and three, then they respond with thirteen, the number of standard deviations would be large, and I could trigger an action.

6.2 Machine Learning-Based Aggregators

Another interesting aggregator would be based on a machine learning model. This would be the most specialized and development-intensive option, but under the correct circumstances could be extremely useful and cutting edge. The pre-requisite for this aggregator is that you have already developed a machine learning model based on data that is being collected by the survey. Then, the model must be hosted in the cloud where it can be accessed by HTTPS. The difficulty with this aggregator is mapping the survey question responses to the inputs for the machine learning model. In addition, there is a lot of flexibility which means there need to be lots of precautions taken from the dashboard to allow all the configuration necessary.

There is a study currently being conducted on TigerAware by Dr. Denis McCarthy that is studying the effects of alcohol use. This study utilizes an "active task" that TigerAware provides: the gait and balance test. This test uses the device's accelerometer sensors to record data while participants walk in a straight line. Dr. McCarthy uses this to gauge impairment of participants.

As a class project, some members of the Lab built machine learning models to analyze the sensor data and predict whether the subject was impaired or not. This is a perfect example of an opportunity to use a machine learning aggregator. The model could be stored in the cloud, then accessed via an aggregator to trigger a multitude of options based strictly on raw accelerometer data. If the model is accurate enough, this could be used by researchers to eliminate the need to provide BACtrack sensors for all the participants – saving a significant amount of money.

6.3 Sentiment Analysis Aggregators

The final proposed aggregator attempts to provide insight into free text responses provided by users. Google and Microsoft have very powerful sentiment analysis models that are available via a REST API. This aggregator would take the text response from participants and feed it to the model to receive numerical values about different sentiments available from the text. For example, say a sentiment analysis model returns happiness, sadness, and anger scores in [0,1] for each text input. One way this aggregator could be used is to retrieve certain sentiment scores for a given emotion. This aggregator would take that output dimension as a parameter, then simply return the value. From there, a greater than or less than operator could be used to trigger an action. These models are the most accurate that are publicly available and offer a very easy to use API – making it a great extension of the Conditional Action Engine.

7 Conclusion

The Conditional Action Engine is a fantastic addition to the TigerAware platform. It engages researchers in their study and provides methods to gain insights that were previously impossible to retrieve. This project created the basic system design including the formal definition of what constitutes a conditional action – an integral part of ensuring extensibility and reliability.

This project also adds the ability for researchers to configure conditional actions from the web dashboard. Also, the backend infrastructure was designed, implemented, and tested to guarantee effectiveness for many different types of data and surveys. In addition, re-usable services were created to send emails and SMS. These services can now be utilized by other parts of TigerAware to better engage our users. Lastly, the groundwork was laid for the ability to extend the system and add more advanced features.

References

[1] "TigerAware," 2020. [Online]. Available: https://tigeraware.com. [Accessed 2020].

[2] D. S. &. Y. S. N. Moskowitz, "Ecological momentary assessment: what it is and why it is a method of the future in clinical psychopharmacology," *Journal of Psychiatry & Neuroscience,* vol. 31, no. 1, pp. 13-20, 2006.

[3] "ZenDesk," 2020. [Online]. Available: https://support.zendesk.com/hc/en-us. [Accessed 2020].

[4] Jessica Marasco, "Automation conditions and actions reference," ZenDesk, February 2020. [Online]. Available: https://support.zendesk.com/hc/en-us/articles/115015611667-Automation-conditions-and-actions-reference. [Accessed 2020].

[5] "Firebase," Google, 2020. [Online]. Available: https://firebase.google.com/. [Accessed 2020].

[6] "RxJS," 2020. [Online]. Available: https://rxjs-dev.firebaseapp.com/. [Accessed 2020].

[7] "Angular Components," Google, 2020. [Online]. Available: https://material.angular.io/components/categories. [Accessed 2020].

[8] "Angular Material," Google, 2020. [Online]. [Accessed 2020].

[9] "Angular Reactive Forms," Google, 2020. [Online]. Available: https://angular.io/guide/reactive-forms. [Accessed 2020].

[10] "Reactive form validation," Google, 2020. [Online]. Available: https://angular.io/guide/form-validation#reactive-form-validation. [Accessed 2020].

[11] "Cloud Functions for Firebase," Google, 2020. [Online]. Available: https://firebase.google.com/docs/functions. [Accessed 2020].

[12] "Firebase Pricing," Google, 2020. [Online]. Available: https://firebase.google.com/pricing. [Accessed 2020].

[13] "NodeMailer," 2020. [Online]. Available: https://nodemailer.com/about/. [Accessed 2020].

[14] "Environment configuration," Google, 2020. [Online]. Available: https://firebase.google.com/docs/functions/config-env. [Accessed 2020].

[15] "Sending Messages," Twilio, 2020. [Online]. Available: https://www.twilio.com/docs/sms/send-messages. [Accessed 2020].

[16] "SMS pricing," Twilio, 2020. [Online]. Available: https://www.twilio.com/sms/pricing/us. [Accessed 2020].

Appendix

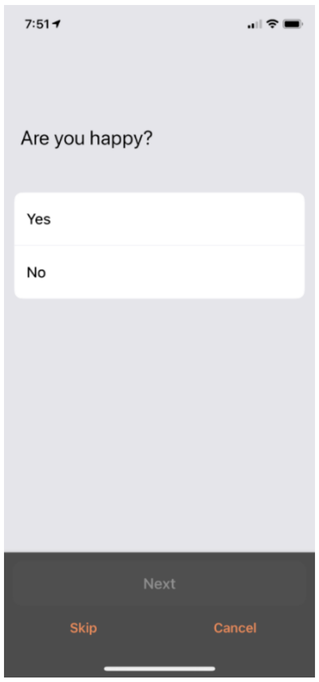


Figure 18: Yes/no question type

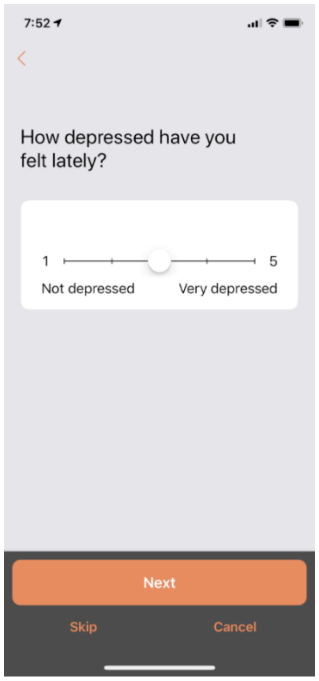


Figure 19: Scale question type

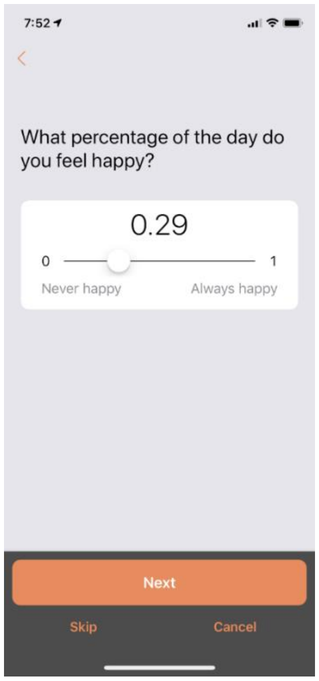


Figure 20: Continuous scale question type

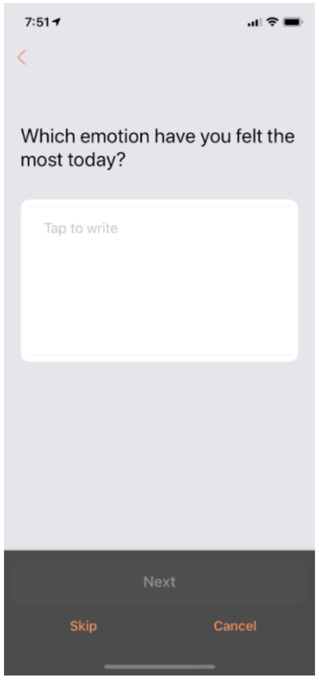


Figure 21: Text field question type

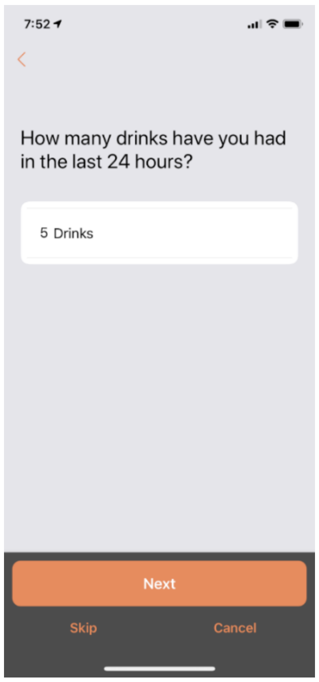


Figure 22: Number question type

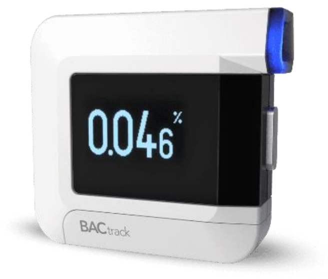


Figure 23: BACtrack sensor used in BAC question type