ALRITE Customization: Arbitrary Data Collection

Ștefan Todoran
University of Washington
Seattle, Washington
stodoran@uw.edu

Navkiran Nijjar
University of Washington
Seattle, Washington
nav1123@uw.edu

Faith Nehren
University of Washington
Seattle, Washington
fnehren@uw.edu

Nicholas Bradley
University of Washington
Seattle, Washington
nbradley@uw.edu

Yitong Shan
University of Washington
Seattle, Washington
yitonsh@uw.edu

ABSTRACT

The goal of the project was to create a versatile and flexible system for customizing illness diagnosis workflows in the Ugandan healthcare app ALRITE. This involved building three concrete tools and a significant design element in order to link said tooling. First, we needed to modify the ALRITE android application to be able to take in an arbitrary workflow specified in JSON format and then parse and display that workflow to the user, allowing them to interact with the workflow, collect patient data, and ultimately receive a diagnosis. Second, we needed the Django backend database to store the JSON workflow files for the Android application, as well as build up the patient information database dynamically based on the information the arbitrary workflow is designed to collect. Third, we needed a powerful yet simple editor in which non-technical users could easily create and modify workflows. Finally, to link all three of these technologies we needed a common language in which diagnosis workflows could be expressed. To further this project, we recommend the integration of our backend services with OpenMRS, the completion of the Android application refactoring we started, the addition of more custom component types, and most importantly the implementation on the Android app of DiagnosisLogic components. During the design and development process, we grappled with poorly maintained code in the project we were building on, weighed the trade-off between simplicity and flexibility when creating a language for expressing abstract branching logic, and contended with the challenges of coordinating work on a three front system with just five developers.

KEYWORDS

ALRITE, customization, digital workflow editor, JSON, Android Studio, Django, data collection, eHealth, TypeScript, data kit

ACM Reference Format:
Ștefan Todoran, Navkiran Nijjar, Faith Nehren, Nicholas Bradley, and Yitong Shan. 2023. ALRITE Customization: Arbitrary Data Collection. In *Proceedings of ACM Conference (Capstone'23).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Motivation

In settings with limited medical resources, applications such as ALRITE [1] serve as a way to assist medical professionals in reaching a diagnosis for illnesses that are often misdiagnosed, such as ALRI (acute lower respiratory illness). In our work, we were tasked with designing and building digital infrastructure that would allow doctors and researchers to create and modify a step by step diagnosis workflow with support for branching logic. In the past, work on the ALRITE project had been bottle-necked by the need for any changes to the diagnosis workflow to first go through the developers, then back to the researchers for feedback, then potentially back to the developers again to fix any issues identified by the researchers. The main motivation for cutting the developers out of this cycle was to speed up the iteration cycle significantly, although added benefits include greater prospects for expansion to other illnesses and a lower likelihood of software bugs.

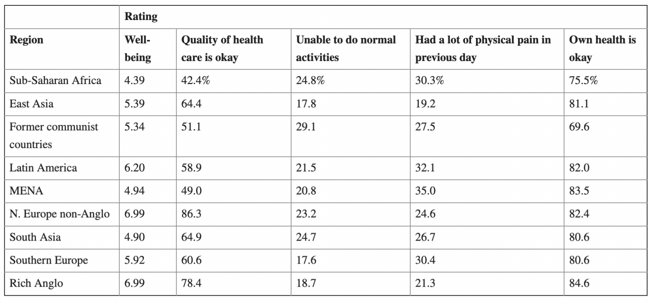
1.2 Background

ALRITE is primarily used in Africa. Below we have attached a figure where the people of Africa rank the healthcare system

Figure 1 shows the low ranking of healthcare in Sub-Saharan Africa [2]. The problem of low healthcare access is what the ALRITE application has been aiming to solve. Compared to East Asia we see an almost 20 percent decrease in well-being. With only 42 percent of Sub-Saharan Africa satisfied with the healthcare they are receiving, ALRITE hopes to lessen that gap and create a way for people in those countries to be able to get better healthcare access where they need it most.

1.3 Overview

The project began with several weeks of meetings in which the existing situation and app were outlined and brainstorming and



	Rating				
Region	Well-being	Quality of health care is okay	Unable to do normal activities	Had a lot of physical pain in previous day	Own health is okay
Sub-Saharan Africa	4.39	42.4%	24.8%	30.3%	75.5%
East Asia	5.39	64.4	17.8	19.2	81.1
Former communist countries	5.34	51.1	29.1	27.5	69.6
Latin America	6.20	58.9	21.5	32.1	82.0
MENA	4.94	49.0	20.8	35.0	83.5
N. Europe non-Anglo	6.99	86.3	23.2	24.6	82.4
South Asia	4.90	64.9	24.7	26.7	80.6
Southern Europe	5.92	60.6	17.6	30.4	80.6
Rich Anglo	6.99	78.4	18.7	21.3	84.6

Figure 1: Peoples ranking of the healthcare system, source: Authors' analysis of data from the Gallup World Poll

design work was done for potential solutions. The design process then gave way to the development process, although design philosophy changes would continue for several weeks. Finally, the last couple of weeks of the project were dedicated to refining the customization system and finishing as many integrations between the various tools (app, server, and editor) as possible.

1.4 Original Roadmap

When beginning the project, the original vision for the workflow editing process involved creating a format to represent the workflows compatible with Excel or CSV editors and allowing the ALRITE researchers modify it there. This approach had multiple benefits, one being that researchers were already experienced in working with Excel, avoiding any learning curve that might come with a custom editor. Additionally, this would not require the development and maintenance of an online editor, which would be a significant undertaking. Finally, this would mitigate the risk of any bugs in said workflow editor impacting or preventing the modification process. Despite these benefits, we found we could not create a well structured and intuitive format to represent the workflows in an Excel sheet, mainly because the nature of a spreadsheet does not work well with the arbitrary numbers of components and pages that would be a part of a workflow. We felt the best user experience would be provided by creating a dedicated editor, and that it was worth the development effort. Despite pivoting to a dedicated editor, we maintained the goals from our original plan, namely that the editing process should be easy and accessible to people with no coding experience, that there should not be a large learning curve, and that the process should not be easily impeded by bugs or issues. A more detailed explanation of our development process for the editor can be found below in the Design Process section.

2 RELATED WORK

2.1 Open Data Kit

One project that solves a similar, albiet more general problem than ALRITE is Open Data Kit (ODK) [3], an open source toolkit for creating data collection apps on Android. ODK offers an online form editor where users can design customized forms with various question types. These forms can then be deployed to mobile devices or accessed via web browsers.

ODK provides offline functionality, allowing data collection in areas with limited or no internet connectivity. Collected data can be stored locally on mobile devices and synchronized with a central server when an internet connection is available. This feature is valuable in resource-constrained areas where reliable internet access may not be readily available. ODK has been widely used in various domains including public health research, environmental monitoring, etc.

Although a relatively old technology, at least by software standards, ODK remains relevant today both directly in terms of its use in data collection and as a case study for the problem of arbitrary/customized data collection.

2.2 Previous ALRITE Work

Upon beginning our work the ALRITE application had built-in workflows which nurses were able to fill out and receive diagnoses related to ALRIs from. The existing workflow was hard coded to a detrimental extent; even if we did not have the goal of creating customizable data collection, more modular code would have significantly sped up the development cycle of the app. The existing workflow, bassed on the IMCI guidlines [4] for ALRIs, had several screens prompting the input of relevant symptoms and support for branching logic based on the entered data. The already existing forms of data input were:

- `Text Input`: The most basic building block of the workflow, used for both numeric and alphabetical input, meaning anything from child weight to parent initials.
- `Multiple Choice`: Used when the input should be restricted to just a handful of possible inputs. Also used when branching logic needed to be done on non-numeric data.
- `Multiple Selection`: Used when input needed to be restricted to just a handful of possible items, but those items aren't mutually exclusive. Also used for branching logic on non-numeric data.
- `Rate Counter`: A technique pioneered by the ALRITE research team for measuring respiratory rate more accurately, this component records the number of taps over a given time interval. Also has support for manual input.

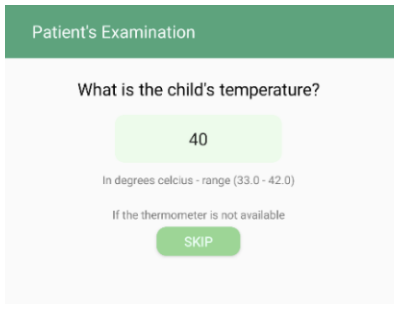
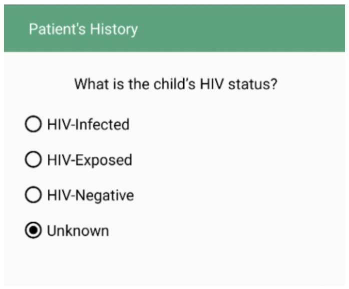

Patient's Examination
What is the child's temperature?
40
In degrees celcius - range (33.0 - 42.0)
If the thermometer is not available
SKIP

Figure 2: Text input example showing a question prompting the user for the child's temperature

The text input component already present in the ALRITE application was in a format similar to figure 2. It consisted of a question with an answer box. The answer box had information about the type of input required, in this case, Celsius. There is also the option to skip the question. Skipping would not alter the decision tree and would allow the user to skip the question under a specific condition. In this case, the skip button was accounting for the possibility that one did not have a thermometer present.

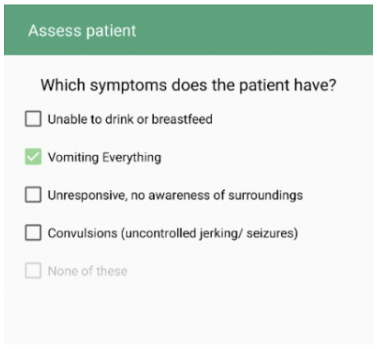

Patient's History

What is the child's HIV status?

- HIV-Infected
- HIV-Exposed
- HIV-Negative
- [✓ Unknown]

Figure 3: Multiple Choice example showing a question prompting the user for the child's HIV status

The multiple choice fragment consisted of radio button logic, whereby The user was presented with a question and then needed to select an singular choice. Similar to the text input, this component always provided an "unknown" answer choice serving a similar purpose to the skip button.


Assess patient

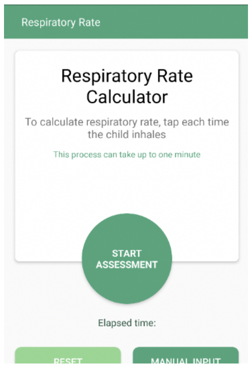
Which symptoms does the patient have?

- Unable to drink or breastfeed
- [✓ Vomiting Everything]
- Unresponsive, no awareness of surroundings
- Convulsions (uncontrolled jerking/ seizures)
- None of these

Figure 4: Multiple Selection example showing a question prompting the user for the child's symptoms

The multiple selection component was originally separate from the multiple choice and served as a way to prompt for questions where there may be multiple simultaneously true answer choices. Similar to the skip button and "unknown" answer choice, this fragment also contained a "None of these" option allowing the user to skip over the question if none of the choices apply.

The respiratory calculator is an interactive input. The user will initiate a timer, during which every press of the button is counted. If pressed every time the child breathes, the component can be used to determine the breaths per minute respiratory rate. At the end, the user would then receive a popup indicating the child's breathing rate and whether it was normal or not.


Respiratory Rate

Respiratory Rate Calculator

To calculate respiratory rate, tap each time the child inhales

This process can take up to one minute

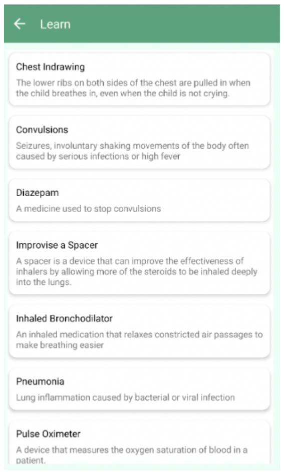
Elapsed time:

Figure 5: A respiratory counter which serves as a button to be tapped each time the child breathes

When the user finished answering all questions they would be prompted with a diagnosis page detailing instructions on how to care for the patient and all of the accrued diagnoses.

Another key feature of the existing ALRITE application was the learn section. The Learn section can be navigated to by users in order to obtain more information on medical terms and possibly learn more about the relevant illnesses.

The existing work on the ALRITE application also included various non-input components, such as image and video components for helping to identify symptoms and demonstrating the administration of care. Another exiting component type was the popup modal, which often appeared in the workflow as a result of logic based on user input.


Learn

Chest Indrawing
The lower ribs on both sides of the chest are pulled in when the child breathes in, even when the child is not crying.

Convulsions
Seizures, involuntary shaking movements of the body often caused by serious infections or high fever

Diazepam
A medicine used to stop convulsions

Improvise a Spacer
A spacer is a device that can improve the effectiveness of inhalers by allowing more of the steroids to be inhaled deeply into the lungs.

Inhaled Bronchodilator
An inhaled medication that relaxes constricted air passages to make breathing easier

Pneumonia
Lung inflammation caused by bacterial or viral infection

Pulse Oximeter
A device that measures the oxygen saturation of blood in a patient.

Figure 6: Learn page of the application including key terms and descriptions.

3 DESIGN PROCESS

The design process of the customization extension was a significant undertaking which required that many details of the workflow format be specified before serious work could be done on the front and back ends of the system. There were also serious design considerations to be made for the Django server and the workflow editor. We first focused on nailing down the workflow format before designing the other pieces of the system. As for the Android frontend, there were certain parts of the existing fragments that would be difficult to import over to a general template. The design choices involved deciding what to keep and what to remove for our version of the ALRITE templates.

3.1 Workflow Format

Being the common language through which the editor, backend and android frontend coordinate, the workflow format was a critical element of the project. Any change to the format for storing arbitrary workflows (or to the specifications of components and pages therein) would have major ramifications for both the Android app and the supporting backend and editor. We knew that as the project went on, the later any change to the workflow format occurred, the higher the cost would be in wasted development time refactoring old code. While a number of changes had to be made to the workflow data format throughout the development process, the front-loading of this design process saved many hours of refactoring.

Originally, the plan was for the editor to merely be a supporting option for workflow creation and modification, and for the backend to be able to convert any JSON-specified workflow into CSV as well, to allow for editing in Microsoft Excel. The motivation behind this was to not "kick the can down the road". As discussed in the motivation section, the point of no-code customization was for the ALRITE researchers to be able to make modifications to the app without going through the time consuming process of coordinating with the developers for that modification to be coded, and later for bugs in its implementation to be fixed. If the editor has bugs that prevent the researchers from being able to modify the ALRITE app, while the overhead in development time has been reduced (once a feature works in the editor it can be used in any arbitrary workflow, rather than being coded up for each workflow) the process of contacting the developers for minor changes has not been eliminated. However, if workflows could be edited by robust and reliable spreadsheet editors, of which there are a variety, a number of benefits stood to be gained. Firstly, the researchers were already familiar with such technologies. Secondly, these spreadsheet editors have significant communities and companies backing them, meaning that bugs are few and response times are swift. Finally, it would simplify the early stages of the development process, as workflows could be edited and modified early on before the editor is complete, allowing for easier progress on the Android frontend and Django backend.

Ultimately, the plan for CSV editing of workflows was scrapped. The primary issue preventing this from being possible is that spreadsheets do not work particularly well for data that has arbitrary amounts of nesting which is not known in advance, nor do spreadsheets accommodate highly sparse data very well. The number of components on any given page is not known in advance. It may be just one component, or it could be several. If each page is represented as a spreadsheet row, this poses a challenge. How can the columns be set up such that any components can exist on any page and in any order? One potential option might be to have an "indefinitely" repeating sequence in the columns whereby the first column is the first component type, the second is the second, and so on and so forth for every component type until we have the first again. If a page does not include the first component type, that column will be empty in that page's row. This however not only would lead to the issue of an incredibly sparse and difficult to edit spreadsheet, but also glosses over the fact that a component cannot be specified within a single spreadsheet cell. While it was for these reasons that we abandoned the notion that the editor should retain CSV editing as a fail-safe, later feedback from the researchers would reveal that the spreadsheet would not have been a desirable editor regardless; many of the complaints and pain points identified about the custom editor by the researchers pertained to it not being clear how to fill out the property fields of components. While the custom editor we developed ended up dealing with these issues in a variety of ways, from tooltip hints to dropdowns with possible values, many of these would not have been feasible or effective in a spreadsheet editor.

One change to the workflow format that came partway into the development process was the scraping of modals. Luckily, no work had been done on modals on the frontend when this decision was made. The motivation behind this was that modals add significant complexity to both the editor and android frontend for something that is ultimately stylistic and not functional in nature. Any content contained in a modal could just as easily be expressed on another page, the only difference being that the modal is visual different from a page. However, the extra complexity this adds to the frontend and editor significantly increases the risk of bugs.

The workflow format settled on is one we believe to be the simplest and most minimal format allowing for the specification of the existing ALRITE diagnosis workflow. Every page which existed in the IMCI diagnosis workflow at the start of this project can be described in the workflow format we designed, with the only exception being that modals are instead represented by their own separate pages.

3.2 Editor

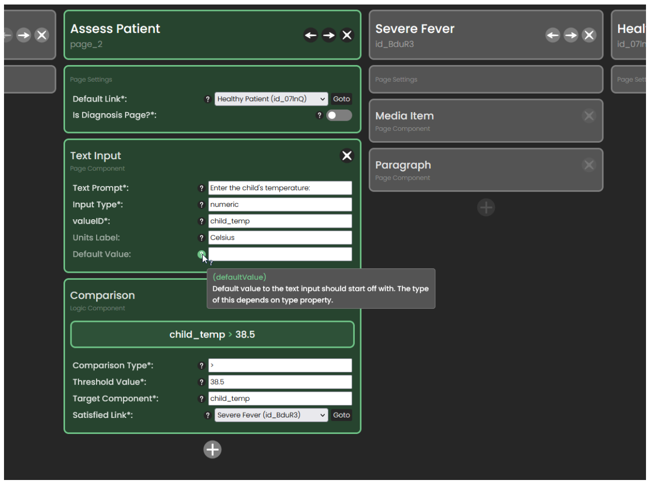
The number one priority with the editor was to keep the design simple, intuitive, and accessible without sacrificing potency. Editing software for technical use cases often succumbs to "menu hell" due to the attempt at jamming in every possible feature and setting. While time restrictions for our project would have never allowed for things to get this out of hand regardless of design philosophy, the goal was to keep things as bare bones and straightforward as possible.

The basic format of the editor is a horizontally scrolling layout containing all pages. Each page displays some basic information as well as its components, but in collapsed form. It is only when a page is selected that it and all of its components expand to reveal their data in full detail. This allows for the user to quickly ascertain a general understanding of the content of any workflow, without there being too much information on the screen at any given time.

The separation of horizontal scrolling being for inter-page traversal and vertical scrolling being maintained for intra-page traversal of a page's components was a deliberate choice in order to keep these two editing actions distinct.

Although the left to right one dimensional layout of pages is not able to fully visually communicate any arbitrary branching logic, something which would require two dimensions, we believe the simplicity of the approach makes the trade-off worthwhile. One is able to immediately and intuitively understand that pages farther right tend to come after those to the left. While with fairly complex branching logic a situation can arise where no one dimensional layout can possibly follow this tendency, there are aspects of the editor designed to address this case. For instance, through the use of "goto buttons" the end user can simulate a specific run through following one branch of the workflow, without the need to ever push changes to the Android app.

Another conscious design decision we would like to highlight is the fairly monochromatic theme of the editor. An extremely powerful technique in UX design is the association of certain colors with certain actions or ideas. One concrete example in the editor is that all elements requiring user input are white or offwhite/super light gray. When contrasted with the green card background for pages and components, in particular in the dark theme, this immediately draws the user's attention.



Assess Patient
page_2

Page Settings
Default Link*: Healthy Patient (id_07lnQ) Goto
Is Diagnosis Page?*:

Text Input
Page Component
Text Prompt*: Enter the child's temperature:
Input Type*: numeric
valueID*: child_temp
Units Label: Celsius
Default Value:

(defaultValue)
Default value to the text input should start off with. The type of this depends on type property.

Comparison
Logic Component
child_temp > 38.5
Comparison Type*: >
Threshold Value*: 38.5
Target Component*: child_temp
Satisfied Link*: Severe Fever (id_BduR3) Goto

Severe Fever
id_BduR3
Page Settings
Media Item
Page Component
Paragraph
Page Component

Heal
id_07ln
Page Sett

Figure 7: Workflow editor sample showing basic branching logic based on collected patient temperature data.

An important metric used in the design of the editor was user feedback. This includes both feedback from end users (the ALRITE team) as well as other involved individuals (such as the professor and TA), and was an invaluable part of the design process. Although the overall form of the editor was established fairly early on in the design process, many seemingly more minute details that utlimately have a large effect on usability were honed in with the help of user testing and feedback.

That said, user feedback was sought out in moderation; an important concept in UI and UX development is not to give the end user too much time with the product before it is finished. It is easy for anybody of any technical experience level to get caught up obsessing over inane details, however it is a particularly common trap for those not well versed in HCI concepts. That is to say, the editor design process, while informed by user feedback, was ultimately driven by tried and testing UX design philosophy.

Overall, while the editor finds itself in a well developed position at the time of writing, a myriad of improvements can certainly be envisioned. More discussion of such improvements can be found in the future work section.

3.3 ALRITE app

Our main priority in designing the app was to keep everything similar in appearance to the original, while still implementing a structure that can be populated with the data received from the workflow. Then, one would be able to click through a series of screens that contain the data from the workflow, and allow for branching logic through those paths.

Another priority in our design was to modify the mechanism that was previously being used to collect data from the app and display it on the final screen, and modify it to be able to work in the context of an arbitrary workflow collecting data which is not known at compile time.

3.4 Templates

As mentioned in the prior section, certain design choices were made regarding the components needed in the Android application. Certain components that were present in the existing application were cut in order to make our design generic and adaptable, as well as to ensure a feasible development timeline.

- `Text Popups`: Text popups as shown in Figure 8 below, were an indicator that users would be presented with typically following some kind of branching or diagnosis logic. These components would present diagnoses such as "The child has a fever" or "The child's breathing is normal".
- `Video Popups`: Video popups were used to serve as a guide for users who did not know what the question was referring to. As seen in Figure 9 below, the user would be presented with a text blurb and a visual video example to assist in question parsing.

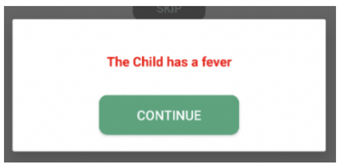


Figure 8: Text popup indicating to the user that the child has a fever.

Neither form of popup was ultimately included in the final version of the project. Instead, it was ensured that through the existing components, any content which would have been placed in a popup

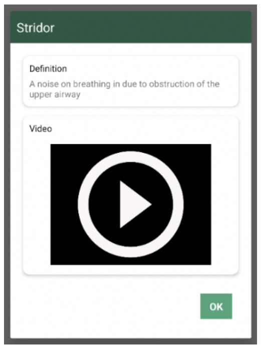

Stridor
Definition
A noise on breathing in due to obstruction of the upper airway
Video
OK

Figure 9: Video popup indicating to the user what the symptoms look like.

modal could be instead placed on its own page. Ultimately, including content in a popup modal as opposed to a page is a purely stylistic choice, which is why the component was dropped.

3.5 Backend

The original ALRITE backend was created using Django, and we decided to continue using Django to maintain as much compatibility as possible. Django also proved to be a good choice due to its support for many different web servers and databases, which made deploying the backend straightforward.

The main addition to the backend was storing and retrieving the workflows, which we chose to do with a new database table. Each entry in the table stores the workflow represented with our JSON workflow format, as well as other metadata and information such as time created. One design choice that we implemented early on was creating a versioning system for the workflows, allowing users to restore older versions and have a record of the changes made. This versioning system was also useful when recording the data collected.

With different workflows present on the backend, the data collection system also had to change dramatically, as each workflow had different questions and diagnoses and thus different data to be collected. In order to handle this, we would have to create a separate database table for each workflow based on the components it contains. What made this more difficult was the fact that we are using Django, which expects all of the tables in the database to be described ahead of time in the source code. These workflow tables would need to be created dynamically at runtime, which sacrifices most of the help that Django provides for managing databases. We originally attempted to create dynamic tables, but we began running into issues with Django caching our models and changes not registering, all issues stemming from the fact that Django expected models to be static at runtime. In the end we decided to use an Entity-Attribute-Value design pattern to store the data, where each datapoint is stored in one large table, and then each entry has a reference to its column name as well as its patient. This way we can store arbitrary data from patients, while still getting the benefits of Django's management. The drawback of this method is it is not as performant, but we felt the stability was worth the small loss in performance.

Integrating the backend with the workflow editor proved to be a vital part of this project, and the area where this was most important was with workflow validation. As we developed the workflow JSON format, we also realized that there were many errors that could occur that needed to be checked for. This validation logic had two priorities: detect common errors and issues in workflows before saving them, and report these errors to the user in an understandable way. Because the validation had to screen all incoming workflows, we decided to put the logic on the backend. This made the second task harder, because we needed to communicate these errors from the backend to the editor. What we decided on was creating validation functions that would return a modified workflow object with only the components that have errors remaining, or as we called it, a workflow validation "artifact". This validation artifact is returned by the APIs on the backend when an error is found, which is then used by the editor to display the errors visually throughout various pages and components of the current workflow.

4 SYSTEM ARCHITECTURE

4.1 Django Backend

The backend is responsible for receiving new workflows from the editor, storing these workflows in a convenient location, supplying the workflows to the Android app, and collecting the resulting data from the app. To make this happen many different changes had to be made to the original ALRITE backend.

4.1.1 Workflow Model. First, in order to store the workflows a new database table was created, represented by a Django model. The fields of the table include: a workflow_id and version field to identify the workflow, a creation time and creation user field to record creation statistics, a JSON field that records information about the changes of the workflow, and finally a Text field that stores the workflow in JSON format. The reason that the workflow is stored in a text field as opposed to a JSON field is because there is no need to automatically serialize/deserialze the workflow, as it is going to be sent to client in this JSON format.

4.1.2 ValueID Model. In addition to the workflow model, we added model to represent a valueID, or the answer to a specific question in a workflow. This model has the folowing fields: A reference to the workflow that it belongs to, a text field for the name, and a text field that records the type of the data recorded.

4.1.3 Patient Model. In order to support arbitrary data collection, the patient model needed to be stripped down. The new fields each patient has are: A reference to the workflow this patient was diagnosed with, the clinician that submitted the record, and the time the record was submitted. These fields are the only common fields that would be shared across workflows, all other fields depend on the workflow that the patient is using.

4.1.4 Value Models. The final models needed are the models that record the actual data from workflows. For each datatype being collected, we have a separate table that stores the specific datatype.

Currently we have a CharValue model and a FloatValue model, and each has the following fields: A reference to the patient that this data value was collected for, a reference to the valueID that this value corresponds to, and the actual value, either a char or float field respectively. With these references, we can create a network between the Value models, Patient models, and ValueID models that represents the data collected from a patient encounter, for any workflow. As mentioned above, this is using the Entity-Attribute-Value method to store data, which has the benefit of being able to record arbitrary

4.1.5 Workflow Validation. An important part of the backend was creating a robust error checker for workflows, as finding and reporting errors in workflows is vital for making the modification process easy and straightforward. This validation logic was placed on the backend so that it can prevent any invalid workflows from entering the database, even if there was a bug in the frontend or similar. The validation logic consists of a set of functions that, when given a workflow object, returns a modified object with only the components and pages that have errors left. This modified object, which we termed an error artifact, can then be sent back to the editor so that proper error messages can be displayed.

4.1.6 Workflow API. With all of the models created, the last step is the API to connect them together. The API has two main endpoints, a GET endpoint that returns the workflow specified by a workflow id and version, and a POST endpoint that creates a new version for a workflow with the provided workflow JSON object. When a new workflow is created, it's JSON object is parsed and checked for errors, and all valueIDs are extracted. If there are errors, the error artifact is send as a response, and no changes are made to the database. Otherwise, a new entry in the Workflow model is created with the provided workflow object. The extracted valueIDs are used to create the necessary entries into the ValueID model, linking to the newly created workflow entry. Finally, a response object is created with relevant information about the workflow, such as its API endpoint for data collection and retrival.

4.1.7 Data Collection API. The data collection API takes in a JSON object that contains all of the valueIDs of a workflow mapped to the collected answer from the workflow. First, a new Patient entry is created for this patient, with the workflow used and the creation time. For each recorded data point, its value is recorded in a new entry into one of the Value models, and it is linked to the valueID that it corresponds to as well as the newly created patient entry.

4.1.8 Dashboard. The final part of the backend is the dashboard, which provides a way for clinicians to look at different workflows, inspect or download the collected data, manage accounts and access, and generally oversee the whole app usage. These dashboard pages were built using Django and Django's templating language, which allows for values from the backend to be substituted into template HTML files. The main pages in the dashboard are: the landing page, which displays some quick stats about the app usage, the workflow page, which lists all workflows with links to edit them or download the collected data, the clinicians page, which lets admins manage users and invite new clinicians, and the patient page, which lists all patients diagnosed with the app along with the data collected.

4.2 Android Frontend

The frontend Android codebase is responsible for displaying the workflow received from the backend, for collecting and displaying user input, and for returning the collected user information to the backend. The codebase is written in Java, and the APK is version 22. There are four main parts to the architecture of the frontend: the workflow collection in the SplashActivity, the assessment in the PatientActivity, the diagnosis screen in the DiagnosisActivity, and the process of sending patient data to the backend.

4.2.1 SplashActivity. This is the activity that is started upon booting up the app. In the beginning, while the app is loading, a splash screen with the ALRITE logo is displayed on the screen. While this screen is being displayed, the app checks if it is connected to the internet or not. If it is not, then it continues as usual. If it is connected, then it will perform a GET request using the Retrofit API to get the JSON data from the ALRITE backend server. It will then store the most updated workflow it receives in the app's data. From here, the app will continue to load as usual.

The Retrofit API was chosen for the request because it is already used in multiple places throughout the project. The app already performs requests to the ALRITE database to collect and validate login credentials, so it makes sense to continue using the same frameworks as the app already has. Even with the existing Retrofit implementation, though, we still needed to add a way to request specifically for the new JSON workflow. To do this, we added the DecisionTreeJSON class, which works with the existing APIClient class to retrieve the data.

4.2.2 PatientActivity: Assessment. This activity is started once the user clicks on the "Assessment" button. Once the activity starts, it checks that the button pressed was the "Assessment" button: if so, then it begins to put together the assessment. First, the app looks for the latest workflow, that was stored upon opening the app by the `SplashActivity`. Then, it loads the file into the app and turns it into a string, which we then turn into a `JSONObject`. From this, we can begin to parse the JSON to create the pages in the assessment. Along with the creation of the `JSONObject`, we also create a `SharedPreferences` object which allows us to store information across the entire activity.

An aside: we have chosen to use the native Java `JSONObject` and `JSONArray` classes to contain and parse the JSON workflows that we receive. This decision was made because of how simple it was to use. While we could have used an external library to handle the JSON data, we aren't doing anything complicated with the workflow: all we do is read and compare the data. Any modifications to this process can be considered by future designers that work on the ALRITE app, if it ends up being helpful.

Once we have turned the JSON into a `JSONObject`, we can begin work on displaying the pages in order. Our method of deciding which page to display begins with determining which type of page we were given to display. This could be a `TextInput`, a `MultipleChoice`, etc. We use an if/else statement within the `getNextPage` function to loop through the possibilities, and once we have a match, we split off and display a different fragment for each type.

We display fragments by replacing the `FrameLayout` in the `activity_patient` layout. Whenever we create a new fragment for a page in the workflow, we create the new fragment of the type that the page is, and then after passing in the necessary information to display the fragment, replace the currently displayed fragment in the `FrameLayout` with our new fragment. We then save the information about the page in a variable within the activity, so that we can reference it later when the user finishes with the page.

Once a fragment has been displayed, we wait for the user to click one of the buttons on screen. If that button is the next button, we are good to collect the information that the client inputted and determine our next steps from that information. We actually don't do that decision-making from inside the fragment: we instead use the Listener model through an interface that was created in the fragment, which calls a function from the `PatientActivity` when one of its methods is called. In this way, we can pass information from the fragment to the main activity. Then, in the activity, we will arrive in the listener function that takes this input.

From this listener function, we can grab the information that we stored in the activity about the current page, and reference it based on the information that was returned from the fragment. For example, if we have a `MultipleChoice` question, and the user chooses option 1 and presses the next button, we can then collect that option 1 and send it up to the patient activity. Then from the activity, we can view the JSON and collect information from any related fields, before storing the result in the `SharedPreferences` object for that assessment, and displaying the next page.

We implement branching logic by collecting information from certain components (logic component, etc.) and determining what to display next based on evaluating from these. For example, with the `TextInput` component we essentially have two different kinds of evaluations that we need to perform: either numeric or non-numeric. If it is numeric, we must implement >, <, >=, <=, and ==. For non-numeric, we really only have to implement == because the rest of those comparisons don't make much sense if it's not numbers. So, for `TextInput`, once we collect the results from the fragment and enter them into the `SharedPreferences` object, we can then check the rest of the `JSONObject` that represents the current page for components with the same `valueID` as the current page. For every component that we find, we check (in sequential order) if we meet the conditions that are specified in that component. If we do meet those conditions, we stop evaluation, find the next page that we should go to, and go to that next page as usual.

One of the considerations that we had to implement newly for the app was the idea of a back stack. Essentially, what should happen if we press the back button on any screen? In the original implementation, the workflow was entirely hard-coded, so the app manually directed the user back to a certain page when the back button was pressed. This would definitely not work for our design, so we had to come up with something different.

We decided upon using a stack to hold the `page_ID`s, as this was the implementation that would be simplest to implement. Once we finished displaying a fragment and collected the information that the user inputted, we would add that `page_ID` to the stack. Then, if the user ever clicked the back button, we would refer to the stack, grab the `page_ID` on top, and then run the `getNextPage` function to re-display that fragment.

4.2.3 DiagnosisActivity. The `DiagnosisActivity` is reached when we reach a page where the `isDiagnosisPage` value is set to true. Once this happens, we store the `SharedPreferences` object in the intent and exit the `PatientActivity`. Then, we create the `xml` page for the `DiagnosisActivity` (which will only display one `xml` page for its runtime) and populate it with the `SharedPreferences` data to display the Summary and Diagnoses sections. We can differentiate between these because of how we stored the data in the `PatientActivity`: the data to be shown in the Summary section has one prefix, while the Diagnoses section has another. Then, the user can click the "Save" or "Save and Complete" buttons (which, right now, do the same thing) and continue on to the `FinalActivity`.

4.2.4 FinalActivity. We didn't modify much in the `FinalActivity`, it still just shows the checkboxes for the doctor's official diagnoses and treatment, and then continues on.

4.2.5 PatientActivity: OtherPatients. The `PatientActivity` doesn't only deal with the assessment: there are three different functions that it fulfills. One of these functions is sending the data to the backend. This is done by entering the `OtherPatients` class, which was pre-existing when we started working on the project. We modified it by making it so that, upon clicking the button at the bottom of the page, it goes through all of the current `SharedPreferences` objects that are associated with a patient. We then collect the Summary map and the Diagnoses list that are required by the backend, and map them to a pair containing the name and version of the workflow that the patient was assessed by. With this, we used the `RxJava` library to batch these requests together and send them at once. If everything goes through, then we can delete the `SharedPreference` objects and refresh the page. Otherwise, we let the user know that something went wrong (probably they weren't connected to internet) and that they should try again.

5 DISCUSSION

5.1 Takeaways

During the 10 weeks, we managed to learn a lot of skills that would be beneficial to our future research and career. Here are our key takeaways from this course and the project:

Technical Skills: We successfully implemented a customizable illness diagnosis workflow system, enabling healthcare workers and researchers to create and modify workflows using the JSON format. We modified the application to accept and display these workflows with different fragments, the app now supports a dynamic number of options for multiple choice questions. We also managed to support reloading the selected questions for the last page when you go back. Additionally, the integration of a Django backend database facilitated data management and retrieval, while we developed a user-friendly web-editor that empowers non-technical users to customize workflows. Although there were challenges starting with poorly-maintained code, we effectively solved the problems and ensured project success. These technical achievements contribute to improving healthcare access, workflow iteration, and software quality in ALRITE. We are also hoping these skills will help us to contribute to future projects.

Group Work: We made a calendar with key dates (prototype due, presentation day, etc.) and group deadlines, such as finishing with the TextInput fragment in 3 days. We listed our progress, issues, and next steps, so we could have quick references every time we need to refer back and keep track of the history. We had regular meetings to sync our progress and do more collaborative work, we had many great discussions and brainstormed for new ideas. We also track our plans and notice each other before key dates. With all of our efforts, we never missed any of the deadlines. The communication between frontend and backend was really important for our team because of how we split our work. We needed to finish our own work but also be familiar with what others are doing. Luckily through good communication practices we were able to work things out.

Meetings with the ALRITE team: We started off with limited knowledge of healthcare in Africa and the target underserved population, we were also unfamiliar with IMCI or any background of ALRITE. We were able to meet with the ALRITE team biweekly, which helped us with understanding their plans and the healthcare situation in Uganda. Other than just completing our tasks technically, we had a deeper understanding of global health and illness diagnosis with WHO and other guidelines, and we are motivated to make more contributions to technology in underdeveloped areas.

5.2 Limitations and Future Work

The most important work to do in the future would be to implement diagnoses within the app. Currently, the app supports collecting information from each fragment. However, we do not have a way to collect, store, and display information that is aggregated and resolved to a diagnosis from across pages. To implement this, one would need to modify the editor and create a new "DiagnosisLogic" type of components similar in nature to LogicComponents but with specific properties relating to diagnoses. Detailed development instructions for such an undertaking are already outlined in the editor GitHub repository, including technical specifications for two types of DiagnosisLogic components. Validation of such components would also be a consideration. Then, within the app, one would need a separate method of accessing and using logic on those components. Such an undertaking would necesitate at least a couple of weeks of time on the backend side to implement this in a way that would not have bugs and would work smoothly with the rest of the application, since there would need to be a way to reference components on different pages from a current page. We decided that, given everything we needed to accomplish in the app, this would be something that we leave for future work on the project, because of the time limitations of the class.

One piece of work that could help with security would be to use an EncryptedSharedPreferences object instead of a SharedPreferences object. If these SharedPreferences objects are stored unencrypted on the device, then it is possible for someone to steal a phone, look through the files, and view unencrypted patient data. To prevent there from being issues regarding this, an EncryptedSharedPreferences object could be used to store the same data, encrypted, on the device. Of course, this could cause the app to slow down and perform worse, but for the issue of security this would be a worthwhile effort.

There can be more types of templates added to the application. One that was considered was a slider. This would be an additional text input that would have the user slide to select a number from a range of numbers, rather than manually type it in. Along with this, there is the possibility of separating multiple choice from a simple true or false/ yes or no answer. Although essentially they are the same thing. It would take a replication of the code with 2 radio button inputs instead of 5.

However, likely the first piece of work that will need to be done is the implementation of the components which currently exist in the editor but are not implemented on the frontend, including the Counter and MediaItem. While the Counter will only require modifications to the Android app, specifically some minor adjustments to the existing repiratory rate counter, the implementation of a MediaItem component will also require changes to the Django backend. This is because the vision for the MediaItem is that it has a file name property, referencing an image or video. This piece of media then needs to be uploaded to the Django backend with the same file name. The Android frontend could then query an API on the backend for any MediaItem files it encounters during the workflow parsing, and cache them locally to allow the workflow to funciton in low/no connection settings.

In the design process section it is mentioned that the editor stands to be improved in many ways. One example of such an improvement would be a visualizer for workflow logic. Although the horizontal, one dimensional layout was the product of a deliberate design decision, a method for viewing just the pages and connections between pages of a workflow would be a valuable tool that could allow the editor to "have its cake an eat it to"; the primary shortcoming of the horizontal page layout is that the branching of a workflow is not immediately visible with just a glance, but a logic visualizer could overcome this issue. A simple solution would be to essentially display the information as a graph, where pages are nodes and links are outgoing edges. Clicking a button on a page card in the current editor could jump to that page's node in the branching logic visualizer, and vice versa.

One limitation of the project, currently, is that it can only support one component per page on the Android frontend (i.e. one `TextInput` or `MultipleChoice` per page). For the sake of completing the project within the given time frame, we decided to only support one component per page. Apart from collecting patient data such as name, age, and weight (which we did implement by hard-coding into the app), this was sufficient for the given IMCI workflow that was currently implemented in the ALRITE app. We realize that in the future it may be desired by the ALRITE team to include more than one component per page, especially as a complaint expressed to use by the ALRITE team was that the nurses had to click through to many pages. This, however, is not something that our team of beginner Android developers had any experience with/felt comfortable developing a solution for, so we leave any modifications in this area up to the more knowledgeable future developers. Thankfully, half the work is already done as the editor and backend fully support any number of components per page.

6 CONCLUSION

The main goal over the course of 10 weeks was to enable non-technical users to create, modify and tweak diagnosis workflows in the Android app, and we were able to make huge strides towards this goal. Our fully functioning workflow editor provides an intuitive way to make small or large changes to existing workflows, or build up a workflow from scratch. The Django backend provides error detection and helps manage different workflows and versions with its dashboard. Last but not least, the Android frontend brings it all together and displays the workflows created in the editor to the end user. Despite many challenges, our team demonstrated effective problem-solving skills and collaboration. We are motivated to continue contributing to technology in underdeveloped areas and believe that our project has the potential to improve healthcare access and quality for individuals in resource-constrained settings.

Altogether this project has been an incredible chance to work with the ALRITE team, and we are grateful to have had this opportunity. We hope that as we pass the torch to the ALRITE team, they will be able to take our work, successes and failures, and continue to iterate and improve the ALRITE application, helping serve the people of Uganda and improve health care outcomes for individuals in underdeveloped settings.

REFERENCES

[1] Laura Elizabeth Ellington, Irene Najjingo, Margaret Rosenfeld, James W Stout, Stephanie A Farquhar, Aditya Vashistha, Bridget Nekesa, Zaituni Namiya, Agatha J Kruse, Richard Anderson, et al. Health workers' perspectives of a mobile health tool to improve diagnosis and management of paediatric acute respiratory illnesses in uganda: a qualitative study. *BMJ open*, 11(7):e049708, 2021.

[2] Angus S Deaton and Robert Tortora. People in sub-saharan africa rate their health and health care among the lowest in the world. *Health Affairs*, 34(3):519–527, 2015.

[3] Carl Hartung, Adam Lerer, Yaw Anokwa, Clint Tseng, Waylon Brunette, and Gaetano Borriello. Open data kit: Tools to build information services for developing regions. In *Proceedings of the 4th ACM/IEEE International Conference on Information and Communication Technologies and Development*, ICTD '10, New York, NY, USA, 2010. Association for Computing Machinery.

[4] World Health Organization. Handbook : Imci integrated management of childhood illness. page Previously issued as WHO document WHO/FCH/CAH/00.12, 2005.