Carleton University

Honours Project

Pocket Scantron

Mathieu Schmid

Louis Nel

Fall 2018

Table of contents

ABSTRACT

Professors commonly use Scantron sheet exams to facilitate grading, however the current process is far from perfect. From long waiting queues during busy times, to not knowing which questions a student entered incorrectly, there is a lot of room for improvement. Pocket Scantron is thus proposed to make this grading process more practical by creating a portable application and providing more in-depth results. Professors can use Pocket Scantron to give on-the-spot exam feedback to students and get a breakdown on which questions a student entered incorrectly.

1. Introduction

Here at Carleton University, and across numerous schools in the country, multiple-choice midterms and exams are typically written on Scantron sheets [Figure 1]. This is a special type of form given to the students in which personal information, course information, and exam answers are provided. The goal of Scantron sheets is to speed up, and automate the process of grading multiple-choice exams. Once a professor has received all of the filled in sheets, they simply pass the bundle through a big, old, single-purpose machine, and, shortly after, receive the original bundle and the marks for each student. However, getting these marks back typically takes a far longer time than it should. Additionally, the current Scantron grading system only provides the grade of each student but does not give information on which questions of the exam a student answered incorrectly. With this all, it is very difficult to receive feedback from professors in regards to Scantron exams.

I propose a new, modern, Scantron feedback tool for professors – Pocket Scantron. This will be a mobile iOS application for professors to give on-the-spot grading of a multiple-choice exam. To mark an exam, the professor must simply select the answer sheet they wish to use, take a picture of the exam, and send those to the server for processing. The answer sheets for each exam graded will be provided by the professor, and stored securely on the device. Once the image has been processed, the professor will be able to see the grade for the given exam, and see which questions the student answered incorrectly.

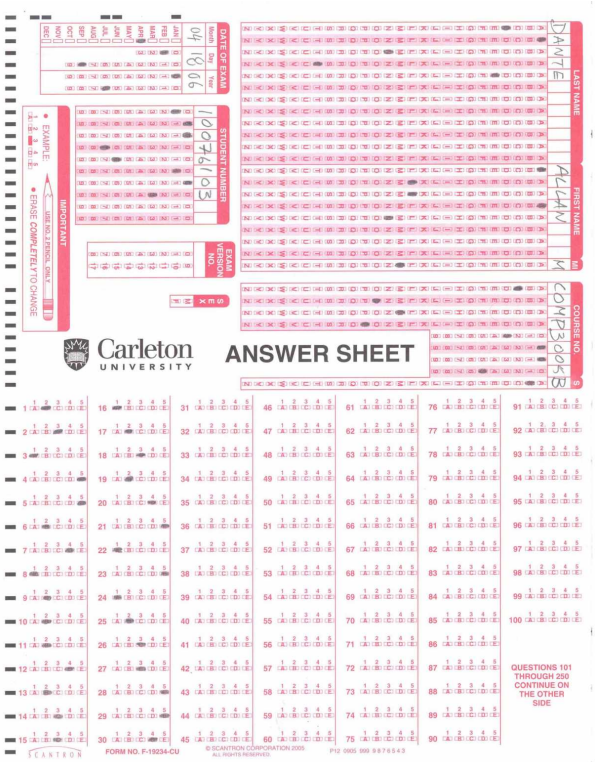

Figure 1. An example Scantron sheet.

In doing this project, I hope many students can profit from the quick and supplementary feedback they can receive on Scantron exams. Students will now be able to go to their professor with any questions regarding their exam, and have on-the-spot details on where they can improve in the course. Additionally, during peak exam periods, professors using Pocket Scantron will be able to skip the queue and grade exams immediately after they are written.

2. Design

When thinking about designing an on-the-go Scantron grader, there are many factors to consider. First, let's consider the natural flow of the app.

1. Open the app
2. If the solutions for the exam-to-be-graded aren't already saved on the device:
    a. Input the solutions to the device
3. Once ready to scan the exam, select the solution sheet to be used when grading
4. Scan the image
5. Send this information to the server to be analyzed
6. When a result response is ready, present view with grade and incorrect answers

The user should be presented with two paths when opening the app; either add a new exam answer sheet, or grade an exam. When adding a new answer sheet, the app will need to take in various exam parameters, such as the name of the exam, the number of questions on the exam (between 1 and 100), and the number of possible answers per question (between 2 and 5). The professor must now input the correct answer for each question on the exam by selecting the correct segment. This answer sheet should be saved on the device to be used later. As these saved answer sheets are highly confidential, they should never be stored on the cloud – only locally on the device. Additionally, professors will also be able to browse and delete their saved exams. If there are no exams saved on the device, there should be instructions to guide the user towards adding one. With this, exam answer sheets are now accessible on the app.

With the desired answer sheet on the device, professors are ready to grade exams. They simply need to select the current exam, and scan the Scantron sheet using the camera built in with the device. The scanned image of the Scantron should be a top-down view to facilitate processing. Thankfully, there exists third party iOS libraries to help with this process – I will use the library WeScan. The user should simply need to take an image of the Scantron sheet from any angle, and crop the image to its bounds. With some magic from the WeScan library, we can capture the top-down, cropped, Scantron sheet image. This image will be sent to the backend and after analyzing the image a response will be sent to the client. For security purposes, the backend will never see the answer sheets, so it will simply take the image as input and output the detected answers for each question back to the client. With the response, the client will then compare each question to

the locally saved answer sheet and formulate an exam grade. The grade, along with any incorrect question will be displayed on a new screen.

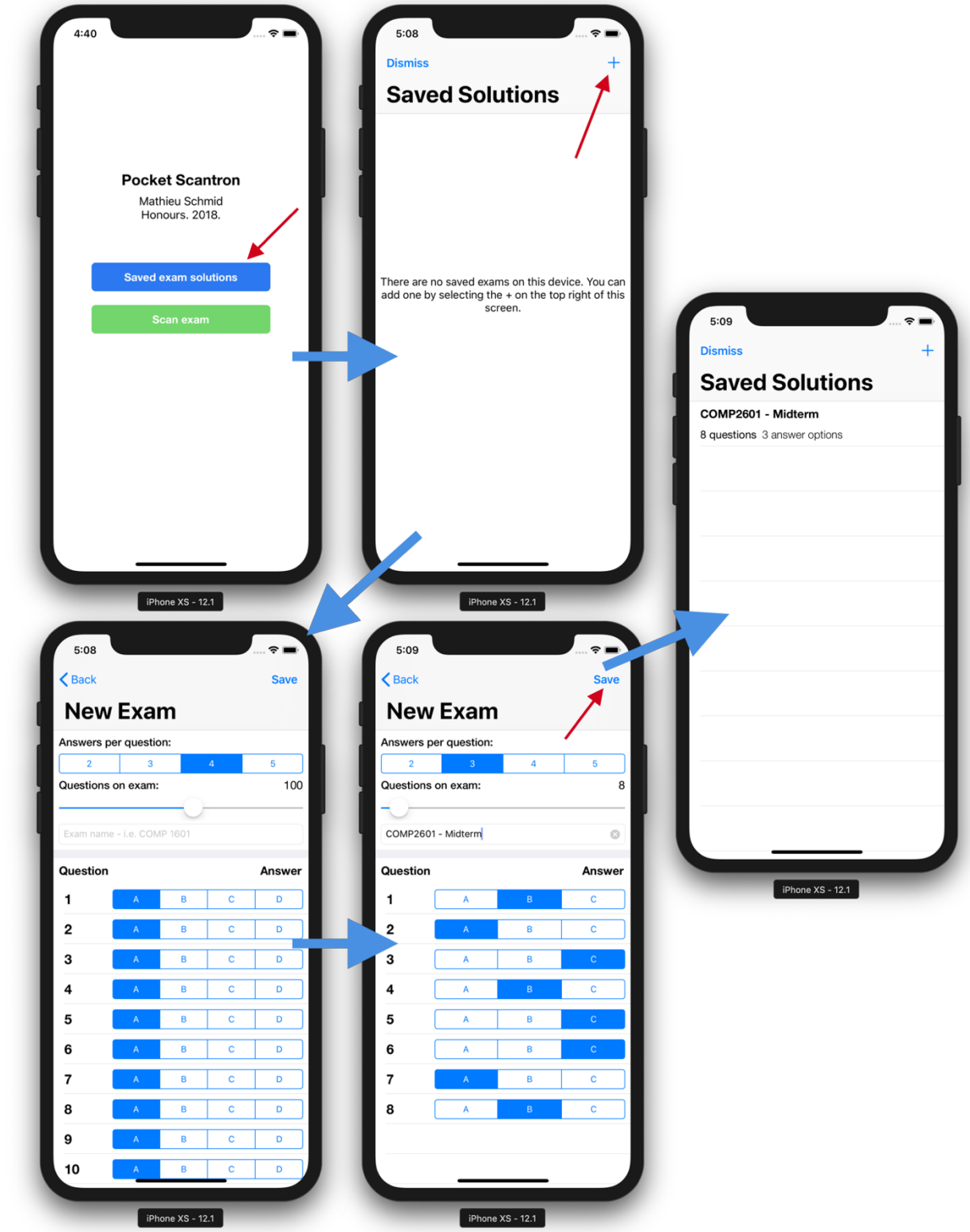

Figure 2. Path for adding an exam solution

The screenshots in Figure 2 illustrate the user's path to adding an exam solution sheet to the device. Every view in this app is built using native UIKit components provided by Apple. It's important to stay consistent with the components provided by Apple in order to have a natural and familiar experience for the users. From the home screen of the app, the user selects the first button "Saved exam solutions" which bring them to the list of answer sheets saved on the device. To add a new exam, the user can tap the "+" button on the top right of the navigation bar. The new exam view contains multiple input fields for retrieving the number of answers per question, the number of question on the exam, the name of the exam, and finally the correct answer for each question. Once completed, tapping save will dismiss this view back to the exam list view, this time with the newly created exam.

Next, scanning and grading an exam. Note: these screens needed to be screenshots from my physical device as the camera functionality is not supported on simulator.

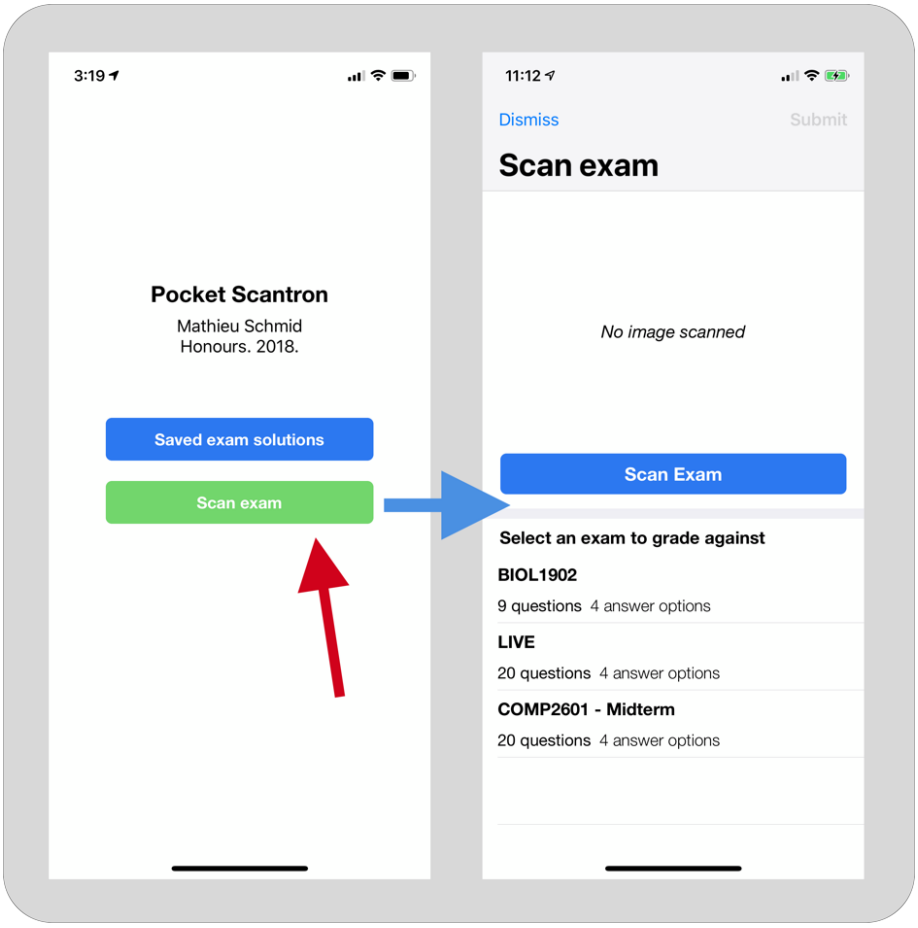

Figure 3. Getting to the Scan exam view

In order to begin grading a Scantron, the user simply selects the green "Scan exam" button on the home screen. This presents the Scan exam view, which collects the exam image and the base exam.

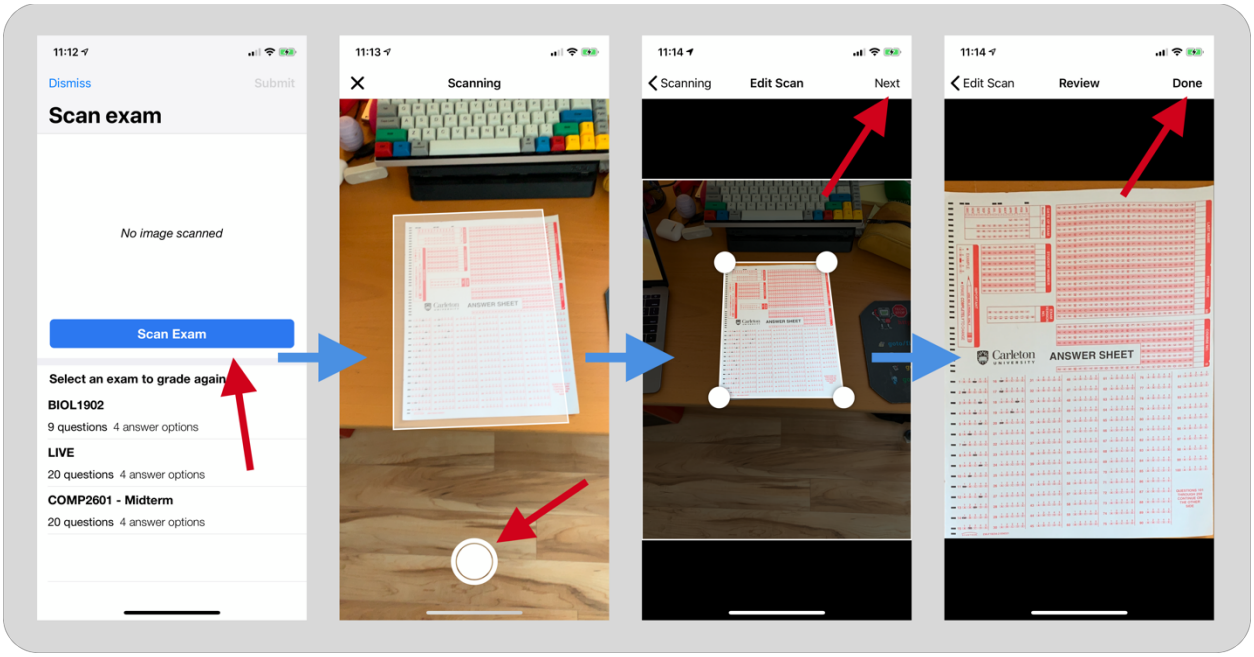

Figure 4. Capturing the exam image

The flow for capturing an exam sheet is meant to be simple and intuitive. Tapping the Scan exam button will open a camera view. All of these image capturing views are part of the WeScan library previously mentioned. The initial camera view provides a rectangular overlay for its best guess at the Scantron sheet within the camera frame. Once an image is taken, WeScan allows the user to adjust the four corners of the Scantron to their nearest approximation of the Scantron sheet. Once the adjustments are completed, the cropped and top-down image is shown. WeScan uses the angle of the surfaces detected to produce the top-down view of the image, allowing users to scan an exam from any angle. This captured image will eventually be sent to the server for analysis.

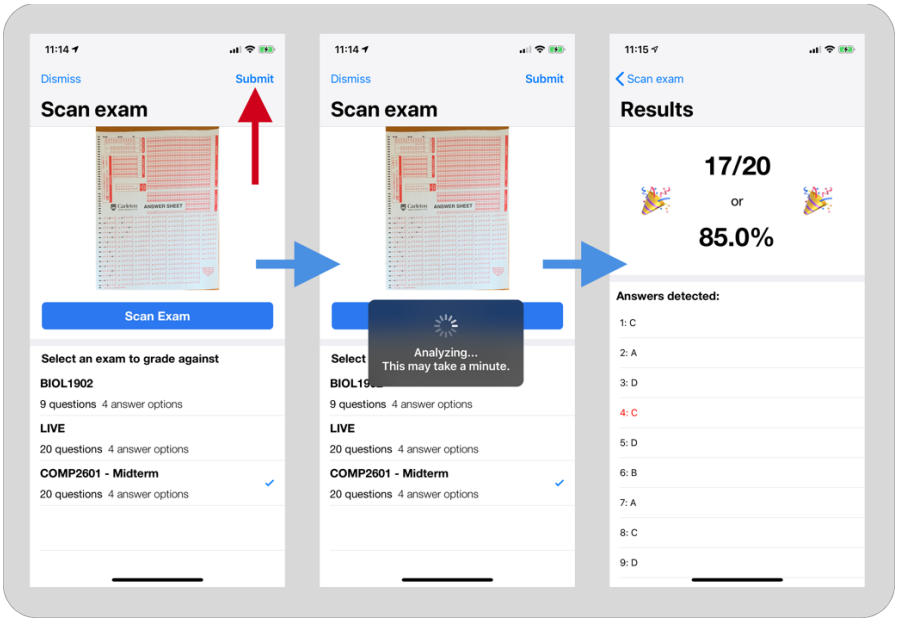

Figure 5. Analysis and results

Once an image scan is taken and a base exam is selected, the professor now has the ability to submit this information for analysis. Tapping the Submit button will start the analysis process. Details on this process will be further discussed later in this report. Once a response is received on the client's device, the results screen is presented with the data. The results screen indicates the student's grade in both a percentage and fraction. There are also two emojis shown on the screen to represents the student's grade**.** The exam answers detected are show in a list, and answers inconsistent with the answer sheet are marked in red. This allows professors to give instant feedback to their students, allowing them to see which questions they got wrong.

3. Implementation

This implementation report will be split into two groups; the front-end report and the back-end report. The front-end report will discuss in detail building the native iOS Pocket Scantron application, whereas the back-end report will focus on the image analysis process.

3.1 Front-end report

The front-end portion of this project represents everything the end user will see and use - in this case, an iOS application. As there are now many frameworks used to develop iOS applications, either natively on non-natively, it is important to clarify this app was built completely natively using the latest version of Swift (4.2) and for the latest iOS firmware (12.1). Building this application natively is not only recommended by Apple, it provides a better user experience (UX) for the users.

Third party libraries were used sparingly throughout, but needed for some fundamental functionality in the app. CocoaPods was the package manager used to handle downloading and versioning all libraries. It is not a tool built by Apple but is unquestionably the most widely used package manager for iOS projects. All of the dependencies used in this project are shown in Figure 6, and each will be elaborated on throughout the report.

```
target 'PocketScantron' do
  use_frameworks!

  pod 'WeScan', '>= 0.9'
  pod 'JGProgressHUD'

  pod 'Firebase/Core'
  pod 'Firebase/Firestore'
  pod 'Firebase/Storage'

  pod 'Alamofire', '~> 4.7'
  pod 'SwiftyJSON', '~> 4.0'

end
```

Figure 6. The entire podfile

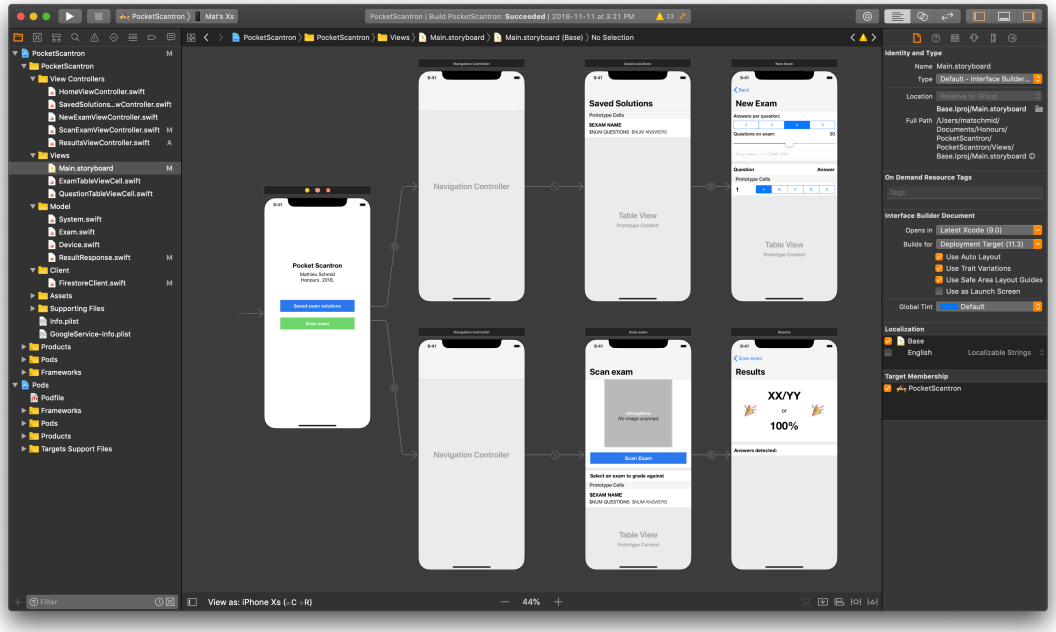

Figure 7. Storyboard file for Pocket Scantron

This application was built in Apple's Xcode IDE, as all of the views in the app were built in interface builder. Interface Builder is an Xcode tool that facilitates building a UI by providing a static representation of each view, where any UI component can be added. Of course, interface builder is not mandatory for iOS building apps, and in fact not recommended when building large-scale apps, but certainly facilitates the process of building a prototype. Interface Builder interacts with `.storyboard` files, which is essentially glorified XML. As seen in Figure 7, an entire app's views can be built in a single storyboard file. It not only represents each view, but also the segue connecting each.

The entire app is a total of 5 unique views and 2 navigation controllers. A navigation controller is part of Apple's UIKit and allows for intuitive navigation between related views. A big part of being a native iOS application is using components which are consistent throughout a user's many applications. Navigation controllers provide fluid and familiar transitions between views, such as swiping from the left of the screen to return to the previous view. In my case, there are two sets of logically grouped views; managing saved exams and grading an exam. Both of these paths can be instantiated from the home screen, which simply provides buttons to each.

As iOS applications traditionally follow a model-view-controller (MVC) design architecture, each unique view is controlled by a "View Controller". View Controllers are responsible for managing the data to be presented on its respective view, as well as handling user triggered events. The third component of MVC, models, are used for representing abstract objects in the app. For instance, in this app there is a model to represent the properties of an exam and a question. Model objects in Swift are typically built from a Struct data type, as opposed to a Class. Structs provide lightweight *pass-by-value* instances of objects, but unlike classes, do not support inheritance. The models representing an exam and a question are shown in Figure 8.

```
struct Exam {
    var id: String
    var name: String
    var questions: [Question]
    var answersPerQuestion: Int
    var numQuestions: Int {
        return questions.count
    }
}

enum Answer: String {
    case A, B, C, D, E
    static let allValues = [A, B, C, D, E]
}

struct Question {
    let number: Int
    var selectedAnswer: Answer = .A
}
```

Figure 8. Struct objects for an Exam and a Question

Now, a deeper dive into each individual view. The `HomeViewController` is the simplest view of the app. It consists of two labels; one displaying the app name and one displaying my name, as well as two buttons. Each of the two buttons represent a navigation path the user might wish to follow.

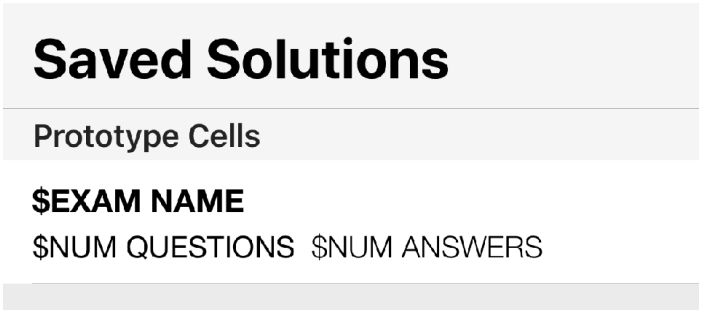

Figure 9. Visual representation of the Exam table view cell

Selecting the blue “Saved exam solutions” button will push a navigation controller housing the `SavedSolutionsTableViewController` view. This view is entirely filled with a list of the exam solutions saved on the device. Apple’s UIKit provides a framework for building list views, coined UITableViews. These table views allow the reuse of cell templates, in this case, each representing a saved exam. The template for an exam cell is shown in Figure 9 and is intended to provide and at-a-glance look at each exam solution. The exam’s name, number of questions, and number of answers per question are shown in this cell. All of this relevant information for each stored exam can be fetched and stored in an array of Exam objects.

From the saved exam screen, if a user taps the top right "plus" button, the `NewExamViewController` is presented. This view is strictly used for gathering user input on every property to do with an exam. Multiple UIKit components are used in this view, such as UISegmentControl, UISlider, and UITextField – these can be seen in Figure 10. A segment control is used for setting the number of answers per question, a slider is used for setting the number of questions on the exam, and a text field is used to get the exam's name. Again, these components should be intuitive to any iOS user as they are used in many system applications. Below these input fields is a table view with a cell for each question on the exam. Each cell contains the question number and a segment control with the correct answer for that respective question. As the slider value is changed, the number of cells on this table view also changes. As cells come in and out of view, the selected answer is maintained. This is done by storing an array with size of the maximum number of questions, but only displaying the first *x* amount, where x is the value on the slider.

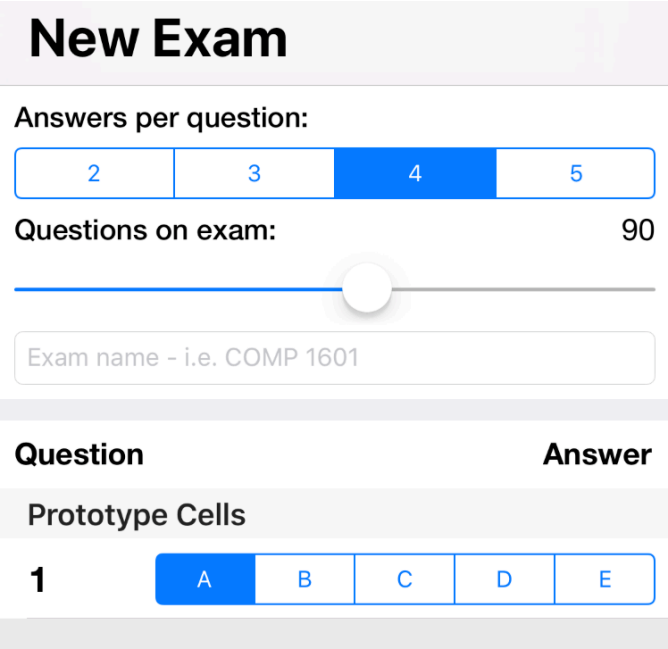

Figure 10. Skeleton for New Exam view

Once a user has completed the input form, tapping the "save" button on the top right will save this exam to the device and return to the saved solutions view. Assuming the user is ready to start grading exams, they can select "Scan exam" from the home screen, which will present the `ScanExamViewController`. This view is responsible for gathering the exam sheet scan and choosing the base exam to grade it against. It consists of a button for opening the camera view, an image view for displaying the capture image, and a table view listing the saved exams on the device. The cells used in this table view are the same as the ones used in the Saved Exams view as they present the same information. As discussed in the Design portion of this paper, the image scanning view is provided by a third-party library "WeScan". This library is extremely beneficial to this project as implementing these features manually would be a huge undertaking. It uses the depth sensors in the device to get angle at which the surface is. With this, a birds-eye-view image

of the exam is very easy to capture. This library also does its best guess at determining the sheet's region in the view. It is generally fairly accurate, but also allows the user to adjust the four corner points for better accuracy. Once an image is captured, it is shown back on the Scan exam view above the button. All that is now left is for the user to select one of the base exams and they are ready to get their image analyzed.

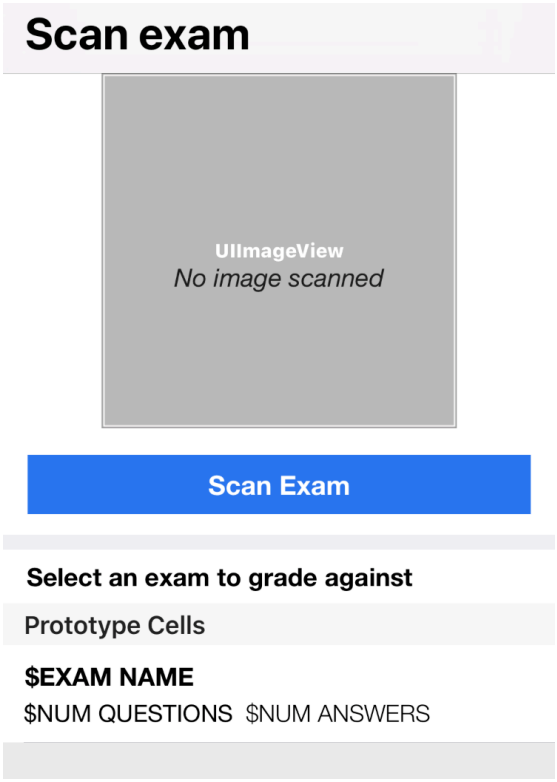

Figure 11. Skeleton UI for the Scan Exam view

Once an image and a base exam is selected, they can tap the top right "Submit" button to proceed with the analysis. With the current technological limitations, this can take upwards of 15 seconds to complete, so a spinner will be shown on the screen in the meantime. A further discussion on what this analysis entails will be explored in the Back-end Report. Once a response is received from the client, the `ResultsViewController` will be shown. This view is responsible for displaying the user's grade and the detected answers for each question scanned. There are two labels for showing the grade in both a fractional and percentage format, as well as two identical emoji labels – as seen in Figure 12. The shown emoji is dynamic with the grade received. Figure 13 is a snippet of the logic for determining which emoji should be shown in relation to the grade received. A switch statement can handle this situation perfectly, but since it's looking at a range of integers and must be exhaustive, a default case must be provided.

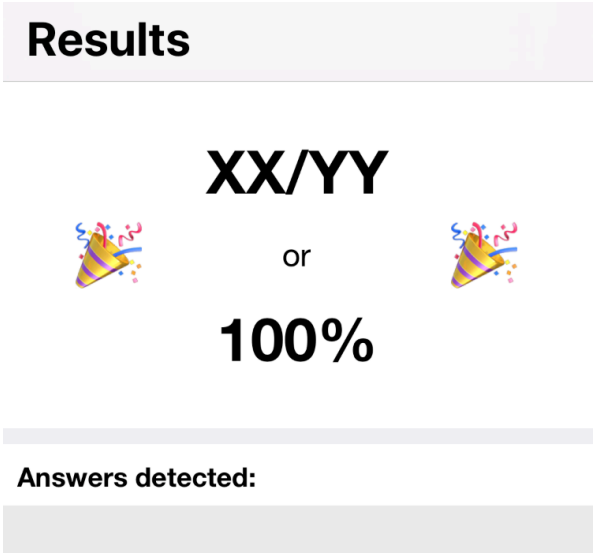

Figure 12. Results view UI skeleton

The exam grade shown on the Results screen is calculated on the front-end end as to not submit any confidential solutions to the cloud. The response from the server contains its prediction for each question on the Scantron sheet. These answers are then compared to the exam selected on the previous screen. With this, all of the information screen is available.

```
switch results.percentage {
case 0..<50:
    emojiLabels.forEach { $0.text = "😵" }
case 50..<65:
    emojiLabels.forEach { $0.text = "😕" }
case 65..<80:
    emojiLabels.forEach { $0.text = "😃" }
case 80..<101:
    emojiLabels.forEach { $0.text = "🎉" }
default:
    emojiLabels.forEach { $0.text = "🚀" }
}
```

Figure 13: Emojis shown on the results view

3.2 Back-end report

This application is not possible without back-end services. The goal of this project is to remove the need for bulky grading machines. This involves gathering and image, analyzing it, then comparing the results to a solution bank. The technology needed for the analysis is not *yet* available in Swift (the language used on the front-end). In order to accomplish this portion of the project, tools only available on different platforms are needed.

The tool and frameworks I used to accomplish this are: Firebase Storage, Firebase Functions, JavaScript, Microsoft's Azure OCR API, Jimp, and TinyImage. With all of these working pieces, it is possible to take an image from the iPhone app, get it processed and analyzed on the cloud, the send the response back to the client.

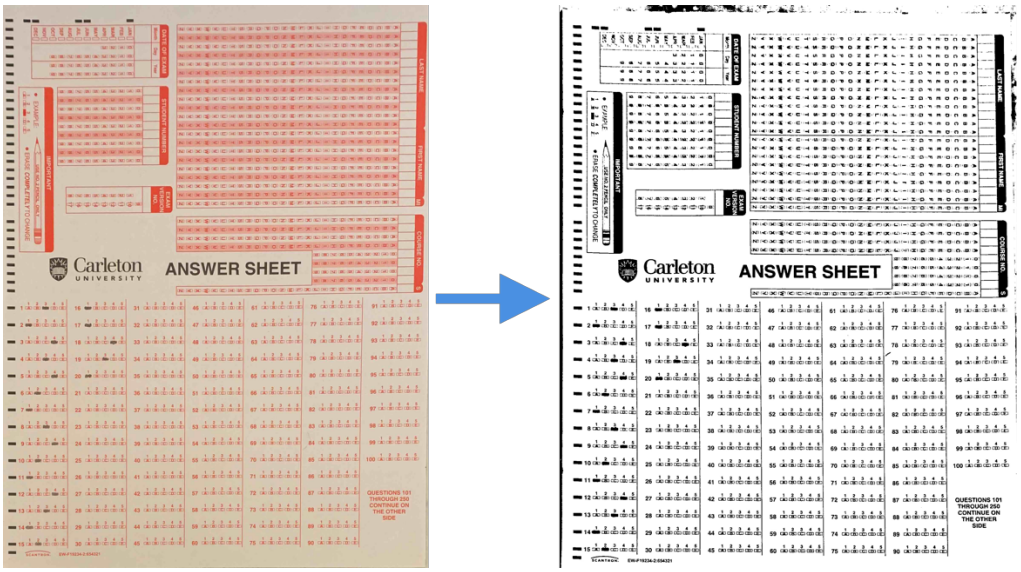

Figure 14. Before and after the contrast and saturation boost

In order to facilitate some server-side work, the contrast and saturation of each captured image were upped directly from the iOS app. Figure 14 illustrates the before and after effect from this image alteration. Having this high contrast image will make the image analysis much easier.

To start the analysis process, the client must first have a way of sending the image to the cloud function. This was done with the help of Firebase Storage, which allows an iOS client to upload an image, and get its URL as a response. For some background on Firebase, it is a *serverless* cloud tool set with a native integration to an iOS application. Its serverless capabilities are what make it appealing to use, as there is very little setup time and cost requirements. With this tool and a simple Swift function, I can easily upload the image capture on the device to Firebase. Figure 15 is a screenshot of Firebase Storage UI, showing the size and download URL to each uploaded image. With the API provided by Firebase, the download URL can be returned to the client as soon as a new image is uploaded.

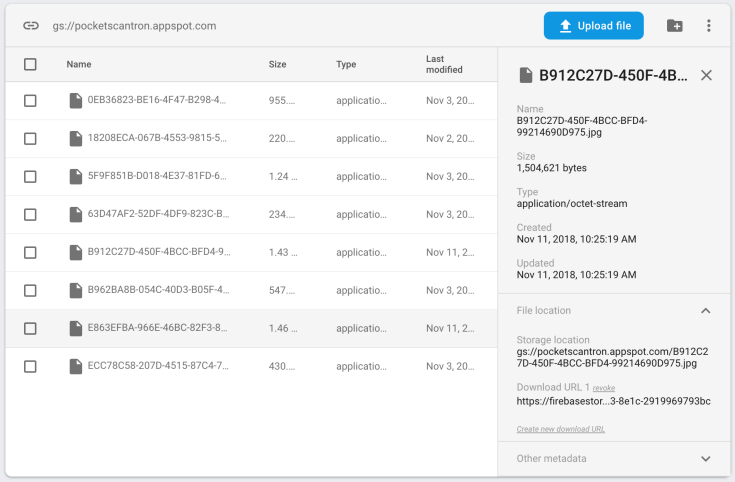

Figure 15. Firebase Storage UI Screenshot

This image download URL can now be used by a cloud function to analyze the image. With Firebase Functions, a REST API endpoint can be hosted and run arbitrary JavaScript code. This ability is extremely powerful for this project, as it allows me to execute JavaScript code from my iOS application. With this, I have setup a Firebase function which takes two parameters as input; the image download URL and the number of questions on the exam. Figure 16 is a screenshot of the Firebase Functions UI, showing I have a method called `documentScore`, its endpoint URL that can be used to access it, and various other meta-data relating to it.

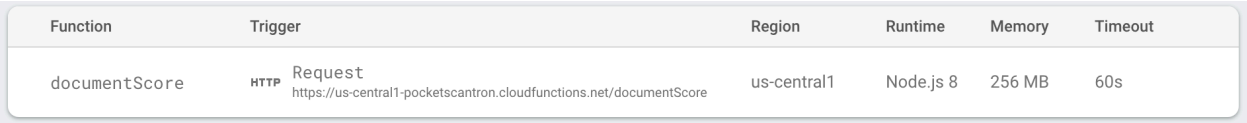

Figure 16. Firebase Functions UI

With this endpoint setup, I can make a network call from my iOS application to this URL with the parameters it needs. There is a very popular Swift library named Alamofire that I use for the networking. Figure 17 is a screenshot of the function I used to make the network call – it builds the URL with the two parameters, makes a network request to it, and receives a JSON response.

```
static func saveImage(url: String, numQuestions: Int, completion: @escaping(_ response: ResultResponse?) -> Void) {
    let url = "https://us-central1-pocketscantron.cloudfunctions.net/documentScore?url=\(url)&numQuestions=\(numQuestions)"
    Alamofire.request(url).responseJSON { response in
        guard let data = response.data else {
            completion(nil)
            return
        }
        let json = JSON(data)
        completion(ResultResponse(withDictionary: json))
    }
}
```

Figure 17. Swift code function to make the network call

Now, to dive into the JavaScript code for this function. The goal of this function is: given a scanned image of a Scantron sheet, return the detected answer for each question on the exam. As only the bottom half of the sheet contains filled in exam answers, we can throw away the top half.

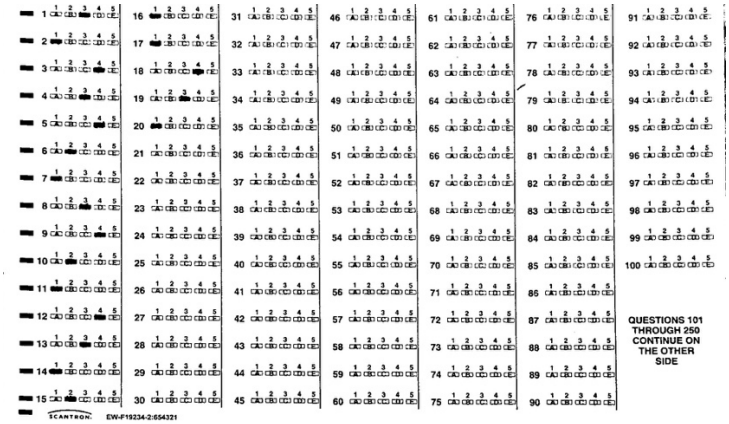

Figure 18. The important portion of a Scantron sheet

Figure 18 illustrates what's left of a scantron sheet after cropping out the top ~55% of the image. Jimp is a JavaScript library that facilitates image manipulation, such as cropping the image. With this remaining image, we can create a grid to properly isolate each question cell. As previously mentioned, Microsoft's Azure OCR library is used. OCR, or Optical Character Recognition, is used to find the bounding boxes of each word or character detected on an image. This tool is extremely useful in this project as it will help us find the most accurate regions for each cell.

Sending the image in Figure 18 to the Azure OCR endpoint will return a whole lot of bounding boxes with the text detected in each. These bounding boxes consist of X and Y coordinates of the origin point (top left corner), as well as the width and height of these regions. Looking at how Scantron sheets are formatted, we can see each cell starts with the question number. If we filter out any bounding box with text that isn't a number, we can find the origin point of each question cell. Unfortunately, numbers 1 through 5 are above each cell, so we'll need to filter those out for relevant results. Numbers 101 and 250 appear in the piece of text on the bottom right of the sheet, so we can filter those out too. What's left is the origin point for question 6 through 100. With the bounding boxes for *most* of the question cells available, we can now begin to build a grid around each cell. In order to form the vertical lines in the grid we can cluster all of the X values for each column on the Scantron sheet (i.e. 1-15, 16-30, …) and calculate the average of those values. Forming the horizontal lines is done similarly, where we cluster all Y values from questions in the same row on the Scantron sheet (i.e. [1, 16, 31, ...], [2, 17, 32, ...]) and calculating the average of those values. Figure 19 illustrates a rough look at the grid we can build following these guidelines.

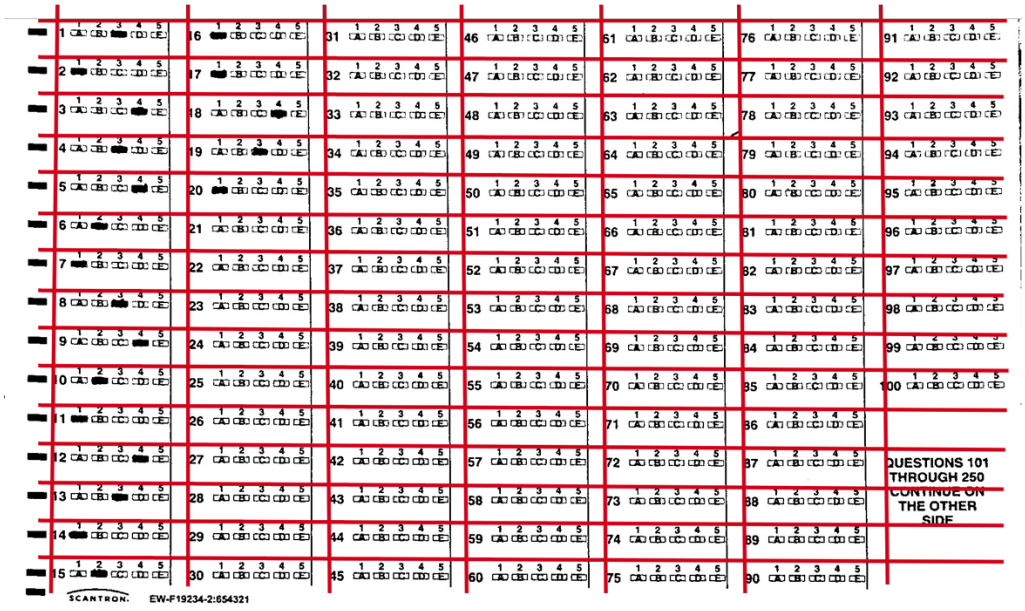

Figure 19. Red lines represent best guess at building question grid

With this grid formed around all of the question regions (or cells), we can "chop" this single image into numerous smaller images, each focusing on a single question. This makes it much

easier to identifying the filled in answer to each question. Figure 20 is three examples of cropped image cells. With a little bit of padding and adjustments, we can build a pretty good image of each cell.

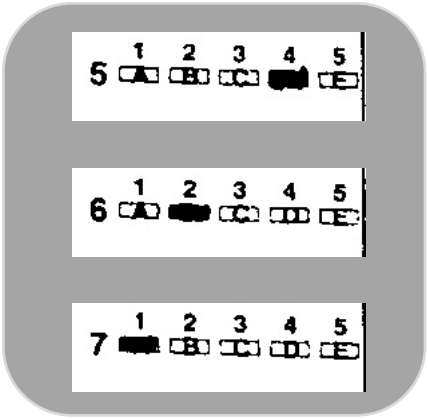

Figure 20. Some cropped image cells

Looking at these cells and seeing which answer is selected is trivial as a human, but not very obvious to do programmatically. The approach I took for determining which answer is selected began with cropping these images ~30px from the left. This was done to remove question number from the images, and thus only leaving the answer boxes. With the remaining image, the next step is to divide it into 5 equal sections along the width – now, each individual bubble is separated.

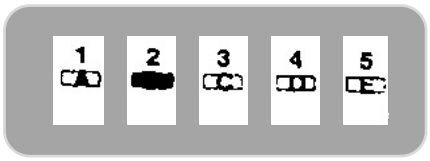

Figure 21. Each bubble is now individually inspected

TinyColor, a JavaScript library, can be used to generate a “luminance” value for a given pixel on an image. The luminance value given is a float between 0 – 1 and approaches a value of 1 as the color resembles white. That is, if the pixel is black, it will receive a luminance value of 0, and as the closer the color approaches white, the higher its luminance value will be. We can use this tool to analyze each pixel on all of the five segments. When summing the luminance value for each pixel on a segment, we can formulate a “total luminance” value for that segment. Presumably, the segment containing the filled-in answer will contain the most “dark” pixels – thus being the segment with the lowest total luminance value.

With this calculation we can make a *fairly* accurate estimation of the selected answer for each question. The function builds a dictionary with a *Question number* to *Selected answer* mapping and stops the analysis process once the dictionary contains the same amount of entries as the number of questions on the exam. This completed dictionary is then sent as a response to the client.

4. Results

With the prototype implementation of Pocket Scantron complete, it was time to start testing it with real images taken from the iOS application. The first couple of tests yielded incorrect results but were easily fixed by adding a little bit of vertical padding to the question cell images. Figure 22 is a screenshot of my environment the first time I was able to receive 100% accurate results with a locally saved Scantron image.

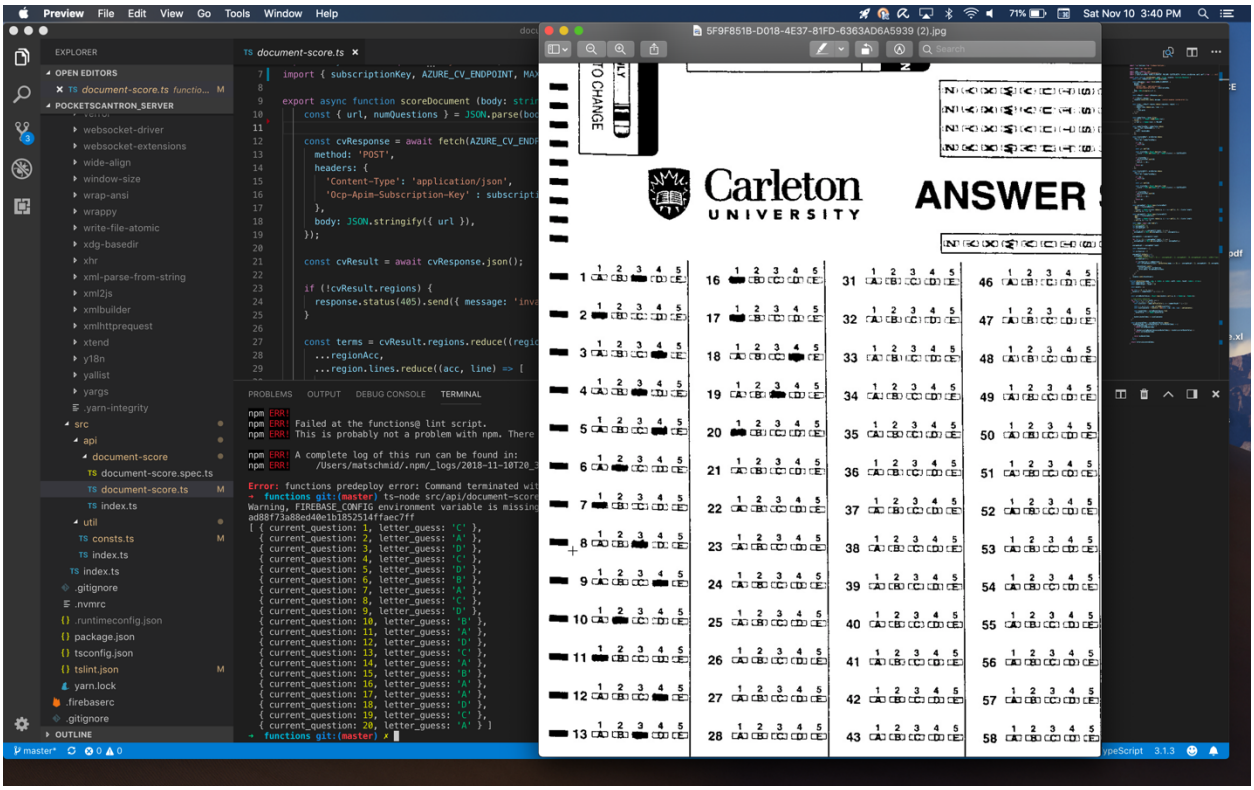

Figure 22. Screenshot from the first time the algorithm worked with 100% accuracy

However, when testing the application on a real device, accurate results didn't seem to happen very often. It is hard to tell why the analysis algorithm worked well locally, but not so well on a real device. One factor might be out of focus or poorly captured images. As this application requires a high quality and perfectly cropped image to work as intended, there is a lot of room for human error. It is hard to measure exact statistics on the analysis accuracy rate of my application, but it unfortunately seems to be in the **20-40%** range. Whether this is related to the images themselves or the analysis algorithm, there is unquestionably room for improvement.

5. Retrospective

With the Pocket Scantron project now complete, I decided to look back and comment on areas I would have done differently. Before starting this project, I had much greater ambitions than what

was technologically feasible. For instance, when writing the proposal, Apple's ARKit 2.0 had just been announced and so I was hoping to have my app use this technology to overlay the exam grade above the exam sheet. However, once I looked into the API and did some preliminary testing, I found out the image recognition technology had a hard time detecting filled in Scantron sheets as they are all unique. Another issue with using ARKit for this project was the time required in the image analysis process. When using augmented reality to overlay results over an object in real time, you would expect them to be shown instantly, but unfortunately this process currently takes upwards to 15 seconds. Had there been an image analysis library available natively in Swift, this project might have been able to use ARKit and perform much faster. One of the reasons for the analysis process taking so long is that it runs from a serverless cloud solution (Firebase). Another reason might be the use of third-party APIs such as Azure – the speed of my service is ultimately throttled by the speed of theirs.

Currently, the primary cause for inaccurate results seems to be correlated with improper image captures. With the currently architecture it is quite tricky to solve this problem. However, one solution to this problem is to build a physical phone mount. This mount would allow the professor to place their phone at a perfect distance above a Scantron sheet frame. This would facilitate capturing the image scan greatly - thus reducing the amount of inaccurate readings.

As this project is meant as a proof of concept, not all of the functionality of a fully working product was implemented. One of the biggest features missing from this prototype is gathering any information from the top half of the Scantron sheet. As shown in Figure 23, this portion includes information like the student's name, their date of birth, the course code, and most importantly, their student number. From a professor's point of view, having a mapping of [*Student number: Grade*] would be extremely useful. This feature would allow them to fully move away from the current Scantron grading system.

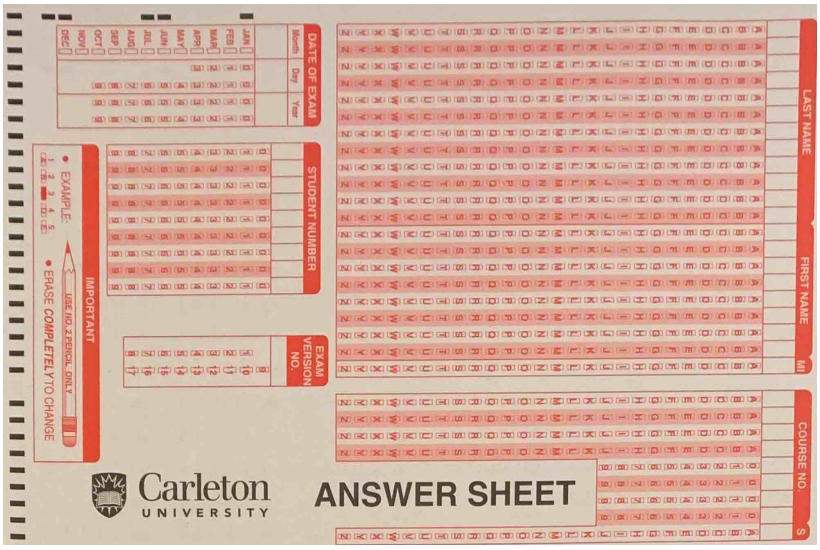

Figure 23. Top half of the Scantron sheet

Another shortcoming of Pocket Scantron is its limitation to 100 exam question. Scantron sheets have 100 questions on the front, and another 150 on the back. In order to support up to 250 questions I would need to extend the image scanning to two image – one from each side of the sheet.

6. Installation and Setup

In order for continued development on this project, it is important to clarify the original setup process and environments used. Along with the implementation section, this section will be split into two parts; front-end and back-end.

6.1 Front-end setup

As the front-end of this project is implemented as an iOS app, Apple enforces a MacOS system environment. I used Xcode 10.1 as the IDE to build this app. Since this client is written in the latest Swift version (4.2) and for the latest iOS firmware (12.1), an Xcode version of 10 or above is required.

CocoaPods is used as a package manager for this project. If it is not already installed on your system, it can be easily added by pasting the following line anywhere in your terminal:

```
$ sudo gem install cocoapods
```

Once CocoaPods is added to your system, it can be used to install and manage Swift dependencies for your projects. In order to install the needed libraries for Pocket Scantron, navigate to the PocketScantron/ directory, and paste the following command:

```
~/PocketScantron $ pod install
```

This should install the necessary dependencies for building and running Pocket Scantron. From here, open the PocketScantron.xcworkspace file from Finder, or from terminal with:

```
~/PocketScantron $ open PocketScantron.xcworkspace
```

This will open an Xcode workspace containing both the Pocket Scantron project and the Pods project which contains all of the source code for the dependencies.

With this, we can build and run the project on either a simulator or physical device. It should be noted that the camera functionality is only available on a physical device, and the plane detection used is only available on an iPhone 6s or newer. The app was designed for an iPhone X / Xs / Xr but should work on any sized device. That's it - the app can now be ran locally!

6.2 Back-end setup

The server-side portion of this project is written in TypeScript, using Node v11.1.0, and deployed on Firebase functions. To get started with modifying the image analysis function, a few basic system requirements are needed. These are Brew, npm, and node.

Brew can be installed using:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

npm can be installed using:

```
$ brew install npm
```

node can be installed using:

```
$ npm install node
```

Finally, from the functions directory, run:

```
~/functions $ npm install
```

From the PocketScantron_Server directory, open its contents in the editor of your choice – I used Microsoft's VSCode. The "meat" of the code can be found in functions/src/api. Here we can find 3 files nested in document-score; document-score.ts, document-score.spec.ts, and index.ts.

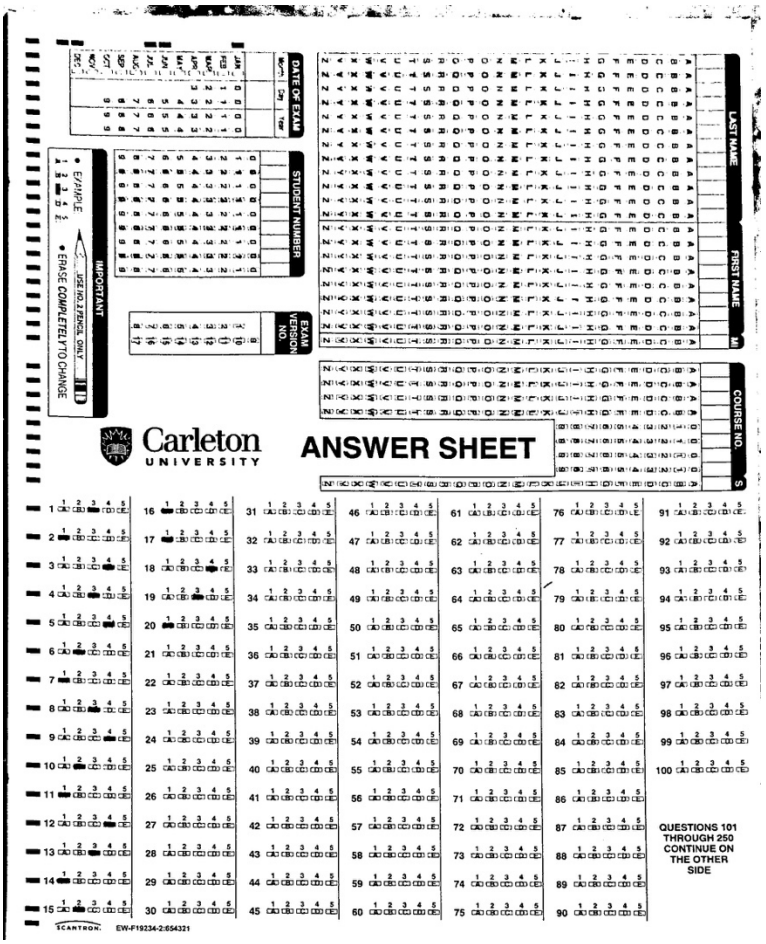

Figure 24. The test image used in the analysis

As the function is deployed to my Firebase functions project, testing the function can only be done locally. The document-score.spec.ts file is setup to run the function locally with a hardcoded test image (seen in Figure 24), and the number of questions set to 20.

In order to locally run this function, use the following terminal command from the functions directory:

```
~/functions $ ts-node src/api/document-score/document-score.spec.ts
```

As seen in Figure 25, the output of the local function is a simple JSON object containing the detected answer for each question.

```
➜  functions git:(master) ts-node src/api/document-score/document-score.spec.ts
Warning, FIREBASE_CONFIG environment variable is missing. Initializing firebase-admin will fail
ad88f73a88ed40e1b1852514ffaec7ff
[ { current_question: 1, letter_guess: 'C' },
  { current_question: 2, letter_guess: 'A' },
  { current_question: 3, letter_guess: 'D' },
  { current_question: 4, letter_guess: 'C' },
  { current_question: 5, letter_guess: 'D' },
  { current_question: 6, letter_guess: 'B' },
  { current_question: 7, letter_guess: 'A' },
  { current_question: 8, letter_guess: 'C' },
  { current_question: 9, letter_guess: 'D' },
  { current_question: 10, letter_guess: 'B' },
  { current_question: 11, letter_guess: 'A' },
  { current_question: 12, letter_guess: 'D' },
  { current_question: 13, letter_guess: 'C' },
  { current_question: 14, letter_guess: 'A' },
  { current_question: 15, letter_guess: 'B' },
  { current_question: 16, letter_guess: 'A' },
  { current_question: 17, letter_guess: 'A' },
  { current_question: 18, letter_guess: 'D' },
  { current_question: 19, letter_guess: 'C' },
  { current_question: 20, letter_guess: 'A' } ]
```

Figure 25. Local function output

In order to deploy this function to Firebase and to be used on the iOS app, access to the Pocket Scantron project is required. Unfortunately, it isn't possible to make the project public, but it would be fairly straightforward to host this function on a new Firebase project.

7. Code Architecture

This section will again be divided into two parts as there are two different architecture approaches used between the front-end and back-end.

7.1 Front-end Architecture

The Pocket Scantron app was built in a very "traditional" iOS manner. MVC, or model-view-controller, was the architectural pattern used. It is known as the traditional pattern in iOS app development as many core UIKit components are built around it, such as UIViewControllers.

Views are what is shown to the end user. There are three options you can use for building views in an iOS app; using interface builder, programmatically, or a combination of programmatic and interface builder. For interface builder you can use either storyboard files or xib files. A storyboard is a single file that comprises multiple static views and defines the segues relating all of them. Xib files are used to visualize a single app view. There is also the option to write your UI programmatically in Swift using the various UIKit components. This method is much more powerful than using solely a storyboard file as there are many more options for customizations. Finally, there is the option to use a combination of interface builder and programmatic layout. This is when the UI components are laid out and placed in interface builder, but the customization of those objects are done programmatically. This is my personal favorite option as you still get the visual representation of your views, but the customization power of programmatic layout. As seen in Figure 26, Pocket Scantron uses the last method of mixing interface builder (specifically, a single storyboard file) and programmatic layout.

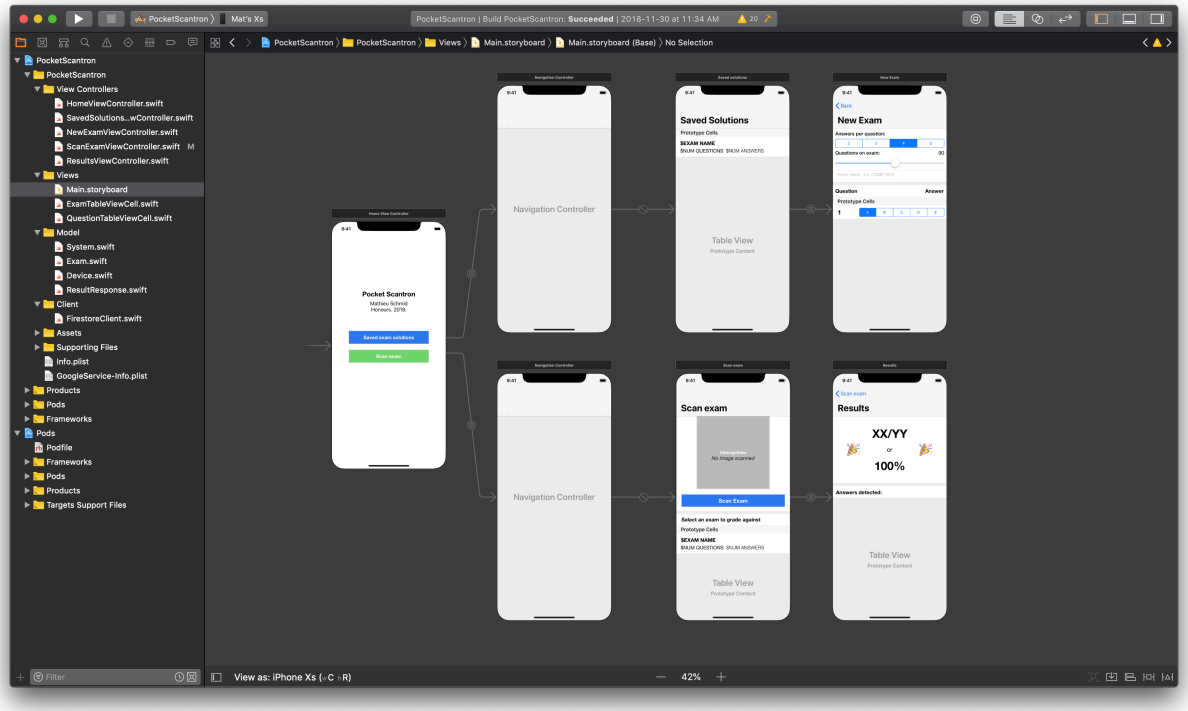

Figure 26. Storyboard file used in Pocket Scantron

Controllers are used to link models with views. In iOS, ViewControllers have the responsibility of managing the content to be displayed on its view and linking the Model classes associated with it. Each view in the app would have its own view controller that are controlled by their respective life cycle methods. Each custom table view cell is also managed by its own view controller. In Pocket Scantron there are five view controllers; home, saved solutions, new exam, scan exam, and results. There are also 2 custom table view cells; exam and question.

From MVC, model classes are used to represent abstract objects. As previously discussed, an example of a model class in the app is Exam. I also have a model class named "ResultResponse" that represents the response received from the Firebase function. Rather than storing the JSON block from the response, it is much cleaner to use a model class to represent it with strongly-typed variables.

Outside of MVC, I have a FirestoreClient class responsible for handling the network request and response from Firebase Functions. This abstracts the networking code from the rest of the app – so, if one day I decide I no longer want to use Firebase functions as my cloud service, I simply need to change this class to accommodate that change.

I also have some helper classes; System and Device. System is used to keep some methods that are used across multiple view controllers. For instance, I have a method that is used to add a label to a table view if the table is empty. There is also a method that simplifies displaying alerts as the code to do so is quite long and often very similar across all alerts. My device class is used to manage storing and retrieving data saved on the device. As discussed earlier, exam solution information is sensitive and cannot be trusted to be stored on external databases, so they are all locally stored on the device. Thankfully, it is pretty easy to do this in Swift. UserDefaults is a tool used in iOS for keeping data locally on the device. For storing, it is as simple as encoding the array of Exam objects to JSON and saving that to UserDefaults. The opposite process is done for retrieving, where you access user defaults and encode the JSON to an array of Exam objects.

7.2 Back-end architecture

The back-end code is separated into two groups; Util and API. Util simply contains a constants file that holds all of the basic information known to the system. For instance, the various padding values and the third-party API keys are saved here. There are also basic functions like getting the x and y value of a given pixel. These are abstracted from the API files in the hopes of "decluttering" them. Nested in API group is the core function logic. The function, named scoreDocument, takes in an image download URL and a number of questions to grade on the exam as input parameters. The function implementation is quite straightforward and doesn't make use of any un-traditional architectural patterns.

It is also important to consider the architecture used to host the cloud function. Firebase Functions is a serverless solution, meaning there is no designated server ready to execute the function. Using a serverless architecture allows developers to focus on building the API endpoints, rather than setting up and maintaining a designated server. It also comes with out-of-the-box scalability, ready for thousands of users. This solution, however, results in slower execution time of the function. When calling the function for the first time, the execution time is between 10-15 seconds, but as it continues to get called, Firebase allocates more and more bandwidth, ultimately resulting in quicker run times. I believe the pros certainly outweigh the cons here – especially when working on a prototype app. If execution time was a bigger factor for the core app functionality, taking the time to maintain a dedicated server would be more worthwhile.

8. Conclusion

Pocket Scantron was an exploration of what can be achieved on a mobile device. The current solution for grading Scantron exams is tedious and not scalable as it requires a large single-purpose machine. In the hopes of building a modern solution for this problem, Pocket Scantron was proposed. It was built as an iOS application for portability and takes advantage of new technologies.

While the original proposal was to overlay the results above the exam using ARKit, various technological limitations prevented that. The final grading flow involves capturing a scan of the exam sheet, uploading it to the cloud for analysis, and comparing the results to a locally stored exam. This flow is far less fluid than the original AR proposal, but is much more stable than the former.

The image analysis performed well locally, but its accuracy significantly decreased when testing on real devices. With accurate results currently only occurring 20-40% of the time, it is unquestionable the algorithm used should be improved. With these statistics, Pocket Scantron should still be seen as a prototype, and should merely be used as a proof of concept. It does, however, have a lot of potential for improvement which can push it to a more trusted and usable state.

In the end, this project has helped me learn a lot about full-stack application development. From building a native iOS application, to writing a server-side image analysis algorithm, this project required a lot of working pieces. Although the final results were undesirable, Pocket Scantron has laid the foundation to creating a better solution for grading exams.