AN INTERACTIVE CROP ANALYSIS PLATFORM FOR IRRIGATION SCHEDULING

A Thesis

by

NICOLAS HUNTER BAIN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Bruce Gooch
Co-Chair of Committee,	Nithya Rajan
Committee Members,	Shinjiro Sueda
Head of Department,	Dilma Da Silva

May 2019

Major Subject: Computer Science

ABSTRACT

Agriculture is a major consumer of freshwater resources around the world. This especially true in Texas, where water withdrawal for agriculture purposes causes the continuous decline of local aquifers. To minimize the effects of this withdrawal and ensure freshwater resources have ample time to replenish themselves, it is important for crop producers to be accurate in their irrigation applications. This research exhibits a web platform that utilizes remote imagery and real-time weather data to generate irrigation recommendations for producers on a field-by-field basis. These recommendations allow producers to accurately gauge how much water their crops require on a daily basis and manage their irrigations throughout the growing season.

DEDICATION

This thesis is dedicated to my loving wife, family, and friends, who continually offer me their support. I am forever grateful.

ACKNOWLEDGEMENTS

I would like to thank Dr. Gooch, Dr. Sueda, Dr. Akleman, and especially Dr. Rajan and her Graduate students, for their guidance and support throughout the course of this research.

I would also like to thank my friends and colleagues for making my time at Texas A&M University a great experience.

CONTRIBUTORS AND FUNDING SOURCES

This work was supervised by a thesis committee consisting of Professor Bruce Gooch [advisor] and Professor Shinjiro Sueda of the Department of Computer Science and Engineering and Professor Nithya Rajan [co-advisor] of the Department of Soil and Crop Sciences

The data utilized in this platform was provided by a number of sources. Weather and agricultural data was provided by the Texas Water Development Board and a weather station operated by Dr. Rajan and her Ph.D. student, Jeff Siegfried of the Department of Soil and Crop Sciences. Landsat data gathered from the Landsat Project, which is a joint initiative between the U.S. Geological Survey and NASA.

Funding Sources

Graduate study was supported by Dr. Bruce Gooch and Dr. Nithya Rajan. This work was also made possible in part by The Water Seed Foundation. Its contents are solely the responsibility of the authors.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

CHAPTER I INTRODUCTION 1

CHAPTER II PLATFORM DESIGN AND REFERENCE ET 3

2.1 Introduction 3
2.2 Framework 3
2.2 Database Schema 6
2.3 Amazon Web Services 9
2.3.1 AWS Overview 9
2.3.2 Configuring AWS for Deployment 13
2.4 Rails Application 17
2.4.1 Rails Configuration 17
2.4.2 Background Tasks 23
2.4.3 Application Flow 31

CHAPTER III REMOTE SENSING 50

3.1 Introduction 50
3.2 Collection and Upload 50
3.3 Landsat Processing 52
3.4 Remote Sensing-based Irrigation Recommendations 58

CHAPTER IV RESULTS AND FINDINGS 61

4.1 ET and Irrigation Recommendation Calculations 61
4.2 NDVI Calculations 61

4.3 User Study 62
4.4 Server Performance 65

CHAPTER V SUMMARY AND CONCLUSION 69

5.1 Summary 69
5.2 Conclusion 70

REFERENCES 71

LIST OF FIGURES

Page

Figure 1. Model-View-Controller Pattern 4

Figure 2. Database Schema Diagram 8

Figure 3. CloudWatch Dashboard 12

Figure 4. CloudWatch CPU Alarm 12

Figure 5. Database.yml and Environment Variables 13

Figure 6. Setup_swap.config 14

Figure 7. Cron.config 15

Figure 8. Yarn.config 16

Figure 9. Nginx_proxy.config 17

Figure 10. Field.rb (field model) 18

Figure 11. Standard HTTP Examples Covered by Resources Tag 19

Figure 12. Route Configuration 20

Figure 13. Fetch_data.rake (1) 25

Figure 14. Fetch_data.rake (2) 26

Figure 15. Fetch_data.rake (3) 26

Figure 16. Fetch_data.rake (4) 27

Figure 17. Fetch_data.rake (5) 28

Figure 18. Fetch_data.rake (6) 29

Figure 19. Calculate_et.rake 30

Figure 20. Ea and Es Calculation Using Humidity 31

Figure 21. Landing Page 32

Figure 22. Station Specific Pages....33
Figure 23. Weather Stations Page....34
Figure 24. HTML.erb Table Generation Code....35
Figure 25. Station_index.js (1)....36
Figure 26. Station_index.js (2)....37
Figure 27. Station_index.js (3)....37
Figure 28. Sign in Page....38
Figure 29. Irrigation Landing Page....39
Figure 30. Field Creation Page....40
Figure 31. Field_create.js (1)....41
Figure 32. Field_create.js (2)....42
Figure 33. Field_create.js (3)....43
Figure 34. My Fields Page....44
Figure 35. Reference ET Recommendation Page....45
Figure 36. Irrigation Controller (1)....46
Figure 37. Irrigation Controller (2)....47
Figure 38. Reference ET Irrigation Recommendations....48
Figure 39. Reference ET Graph....48
Figure 40. Precipitation Graph....49
Figure 41. Field_show.js....49
Figure 42. Landsat Upload Page....51
Figure 43. Progress Bar JavaScript Function....52
Figure 44. Landsat Irrigation Recommendation Page....54
Figure 45. Process_landsat.js (1)....55

Figure 46. Process_landsat.js (2).....56
Figure 47. Jordan Curve Theorem Function 56
Figure 48. Landsat Processing Page.....57
Figure 49. Processed Landsat Display 58
Figure 50. Irrigation Controller (3) 59
Figure 51. Irrigation Controller (4) 60
Figure 52. Task 2 Timing Results 63
Figure 53. Task 4 Timing Results 65
Figure 54. Elastic Beanstalk Load Test Metrics 68

CHAPTER I

INTRODUCTION

While many industries have been quick to adopt technology that incorporates data-driven decision making, agriculture has fallen a bit behind. This is especially true for operational irrigation scheduling in Texas. Current irrigation methodologies in Texas revolve around the **Reference Evapotranspiration (ET)** approach. This approach involves two steps. The first of which is the calculation of a reference ET for a hypothetical reference crop using a modified form of the Penman-Monteith Equation. The second is to calculate crop evapotranspiration (ET_c) by multiplying reference ET by an empirically determined factor, the "crop coefficient (kc)," which is specific to the crop type and agricultural region. ET_c is generally calculated on a daily basis. It represents the amount of water lost from the crop canopy and soil surface, which then needs to be replenished by irrigation (or rain) for the crop to remain in a healthy, well-watered state. The issue with the kc approach is that it calculates ET_c under standard conditions. Unfortunately, fields are rarely in a perfectly ideal state and farmers that rely heavily on this method can end up over-irrigating when conditions for their field limit crop growth or ET.

To amplify irrigation scheduling efficiency, crop water demand needs to be evaluated on a field-by-field basis. It should account for real-time plant and weather conditions. At the moment, there are not any operating systems available to provide this crucial

information to farmers in Texas. They are limited to public weather station data and are rarely provided with the daily values necessary to calculate reference ET.

The best method to obtain the data necessary for these real-time recommendations is to utilize remote image data. This type of imagery has become more available in recent years due to satellites and unmanned aerial systems. With it, we can calculate vegetation indices such as the normalized difference vegetation index (NDVI). Using NDVI, crop coefficients can be adjusted for individual fields. This is possible since the seasonal evolution of remotely sensed NDVI is similar to that of a crop coefficient. Thus, crop coefficients can be modeled as a function of NDVI, which gives us a method for adjusting the crop coefficient values at the field scale for crops such as cotton, corn, wheat, and soybean.

CHAPTER II

PLATFORM DESIGN AND REFERENCE ET

2.1 Introduction

The purpose of this chapter is two-fold. It begins by covering the overall platform design and proceeds to describe how it is utilized to create reference ET-based irrigation calculations on a field-by-field basis. The platform design is broken down into several sections, with each covering a specific component. The first section describes the design and setup of the framework. The second covers the database architecture. This includes the database schema and model relationships. This is followed by a section covering Amazon Web Services and how they were configured to work with the application. The final section covers Rails application itself. Specifically, the configuration, core functionality, and application flow.

In this section, the scope of the application flow overview is limited to reference ET-based irrigation calculations. This is for the sake of simplicity. An additional overview, which covers the utilization of remote sensing data is covered in the following chapter.

2.2 Framework

To implement this system, I chose to utilize a framework that excels in handling database-backed web applications: Ruby on Rails™(RoR or Rails) [2]. Rails is a web framework written in Ruby. It utilizes the Model-View-Controller (MVC) architectural

pattern conceived by Trygve Reenskaug that is commonly seen in user interface applications. In the MVC design pattern (**Fig. 1**), objects in an application are assigned one of three roles and the communication between roles is strictly defined. Each role (model, view, or controller) is separated from the others via abstract boundaries and is only allowed to communicate with objects of the other types across said boundaries.

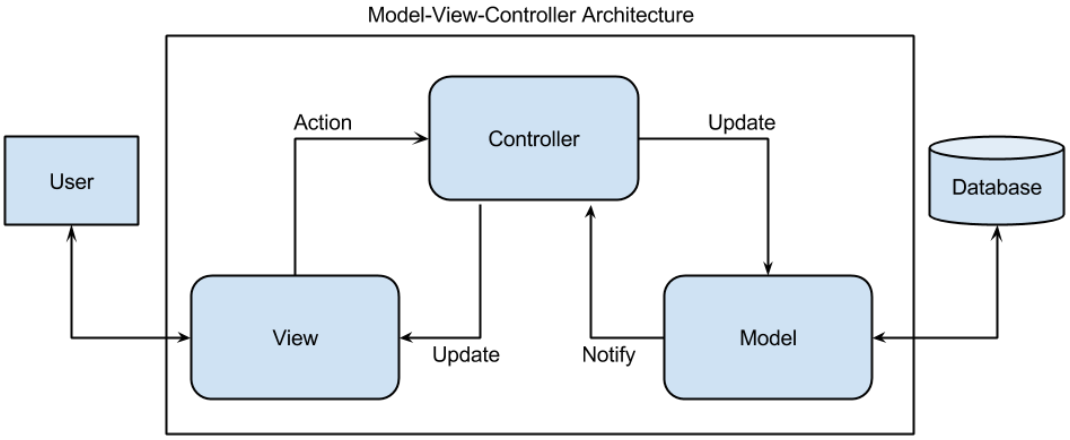


Figure 1. Model-View-Controller Pattern

Reenskaug describes the model role as one designed to represent knowledge. While a model can be a single object, it is usually a structure of objects. “There should be a one-to-one correspondence between the model and its parts on the one hand, and the represented world as perceived by the owner of the model on the other hand [1].” Each node of the model should represent a distinguishable part of the problem and all of them should lie on the same problem level.

A view acts as the visual representation of its corresponding model. Views generally highlight the attributes of a model that users are interested in and suppress those that are

only necessary on the application's back end. Because of this, views are sometimes called presentation filters.

Controllers act as the link between the user and the system. Each one provides the user with input by arranging for relevant views to present themselves in the proper places on the screen. They provide means for user output by presenting the user with menus or other means of giving commands and data. The controller receives this user output, translates it into the appropriate messages, and passes the messages on to one or more of the views.

One of the benefits of Rails is its built-in support for the major database types. The application creates an abstract layer through its ActiveRecord module, which acts as an object-relational mapping system for database access. This offers developers the freedom to choose database types based on need or preference. In the case of this application, PostgreSQL was used due to its ability to recursively input massive amounts of records. Another benefit of Rails is the fact that it is open source. This has resulted in the community-wide development of libraries. These libraries are collections of modules written in Ruby, which can be easily incorporated into new or existing Rails applications. This application makes use of a number of libraries to relieve some of the coding necessary to develop it. For example, it utilizes libraries for geocoding and user authentication.

Another benefit of Rails is designed to emphasize Convention over Configuration (CoC) and the Don't Repeat Yourself (DRY) principle. CoC meant that it was only necessary to specify the unconventional aspects of the application and DRY alleviated the need for some repetitious tasks.

2.2 Database Schema

The core of the application architecture is the relational database. There are several base models with associations between each other and additional sub models. The following page shows a graphical representation of the schema (**Fig. 2**). There are several core models: crops, stations, Landsat, fields, and users. Crops simply have a type (sorghum, cotton, or corn). However, each crop has a sub model associated with it called a KC coefficient. The KC coefficient is the ratio of daily ET observed for a reference crop (ET_c) and the actual ET of a crop. There is a KC coefficient for each day of the crop's growth, which is used in the irrigation recommendation calculations later down the line.

The station model has an instance for each known weather station in Texas, including the Texas A&M Research Farm weather station. The instance includes the station's name, county, latitude, longitude, elevation, and a Boolean value to keep track of whether or not reference ET can be calculated for the station. The station model has two sub models; readings and ETs. Every 15 minutes these stations report on agriculture and

weather conditions for their area, which are logged as instances of the Reading model. Every 24 hours, the readings collected over the past day are utilized to calculate reference ET for that day. These ET values are utilized when generating irrigation recommendations.

The Landsat model contains the latitude and longitude values for each of the corners of the Landsat image and two constants per spectrum (red and near infared) that are necessary in accounting for reflectance in NDVI calculations. In addition to this, it has active storage references to both of the actual spectrum images (red and near infared), which are stored in an Amazon S3 bucket.

The last two core models are the user and field models. The user model directly owns the field model and therein the irrigation recommendation model indirectly. Each user instance contains standard user information such as username, email address, and an encrypted password. The field model includes a name, the id of the crop that's planted there, the planting date, the closest station, and the geometric information for the field. The field model directly owns the pixel set model and therefore indirectly owns the pixel model. Pixel instances include latitude, longitude, and NDVI values for use in remote sensing-based irrigation recommendations.

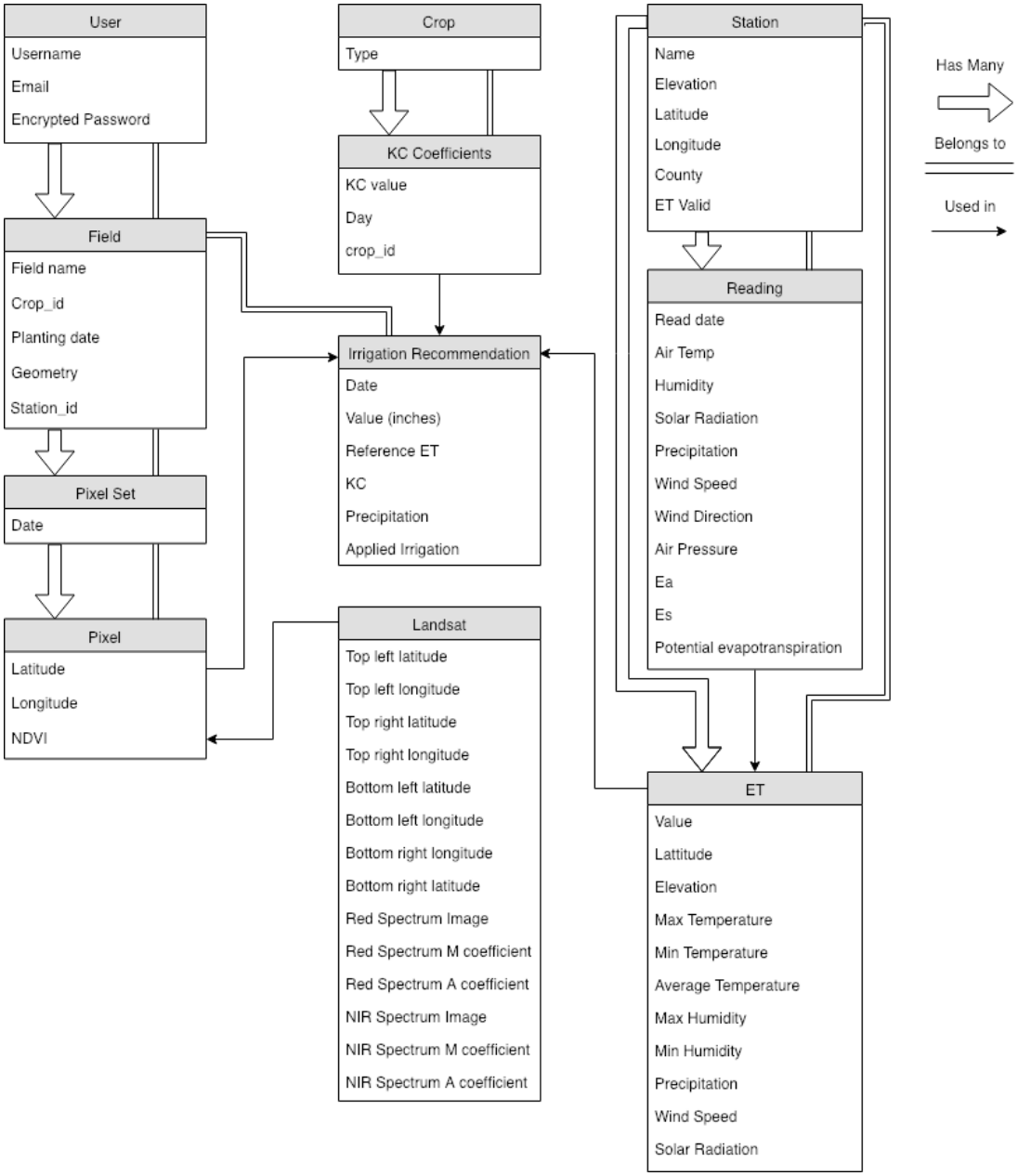


Figure 2. Database Schema Diagram

The final model is the irrigation recommendation. This model utilizes the ET model and either the KC coefficient or pixel model depending which type of irrigation recommendation the user requests. Each instance contains the calculated value, the date, the reference ET, the KC or NDVI value used in its generation, daily precipitation, and total solar radiation for the day. It also has a value called applied irrigation, which defaults to zero and is modified by the user if they wish to keep track of how much they irrigated throughout the season. If a producer desires to practice irrigation banking, this would help them do so. The processes and instance generations will be discussed in more detail in the following sections.

2.3 Amazon Web Services

2.3.1 AWS Overview

There are a number of cloud-based hosting services available today, but in the end, Amazon Web Services (AWS) was the optimal choice [5]. The main reason for this lies in the versatility of their Elastic Beanstalk service. This is an orchestration service that acts as a monitor, manager, and container for AWS's underlying infrastructure, which includes over a hundred different services. With it, I was able to develop the platform in a fashion that supports automatic scaling to meet user demand as platform usage grows. The platform will never need to be relocated or redeployed simply because the hardware configuration could not support an uptick in traffic. This limits the need for maintenance and oversight dramatically. The added bonus of this configuration is that Dr. Rajan and her team will only have to pay for resources that they need, keeping the cost reasonable.

The Elastic Beanstalk service enhances developer productivity while simultaneously offering complete resource control. It provisions and operates the infrastructure and manages the application stack autonomously so the developer doesn't have to. This includes, but is not limited to applying application updates or patches, configuring servers, databases, load balancers, firewalls, and networks. If necessary, developers can go "under the hood," so to speak, to configure all of this directly.

The core elements handled by the Elastic Beanstalk service are the EC2 instances, S3 buckets, Elastic Load Balancers, RDS, and CloudWatch. EC2 refers to Amazon's Elastic Compute Cloud. These are virtual private servers running an Amazon Machine Image (AMI) which is configured based on the user's needs. For example, the EC2 instances used in this application are running Puma with Ruby 2.5 on 64bit Amazon Linux 2.8.4.

S3 buckets refer to Amazon's Simple Storage Service, which provides object storage through web service interfaces. In this case, the Representational State Transfer (REST) architectural style is used to work in tandem with Rails. These "buckets" contain application version archives and any Landsat images that are uploaded to the platform. These images are located and linked to the platform running on the EC2 instance via storage blobs within the Postgres database. This is necessary because of the sheer size of Landsat images. Each image is over 100MB and there are two per Landsat instance.

The Elastic Load Balancer automatically distributes application traffic across multiple targets. If user traffic increases to the point where it's necessary to run multiple EC2 instances to ensure optimal user experience, the load balancer will split traffic and direct it accordingly. This works in tandem with EC2 auto scaling.

RDS refers to Amazon's Relational Database Service. RDS is a cloud-based database service that allows developer to set up, operate, and scale relational databases. "It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups." This application utilizes PostgreSQL on a scalable database instance that is backed up every 24 hours to limit potential data loss.

CloudWatch is the name of the service that provides real-time monitoring of EC2 instance resources. It provides resource utilization metrics for CPU, disk, network, and replica lag for RDS database replicas. It is also the service that enables auto scaling. Developers can create alarms based on thresholds for resources such as the CPU, which will add or remove EC2 instances as needed. This application adds EC2 instances when CPU utilization is above 90% for longer than five minutes and removes them once the utilization drops below 15%. The dashboard for CloudWatch is shown below (**Fig. 3, 4**). Developers can add graphs and alarms for any number of metrics to maintain full control over their application.

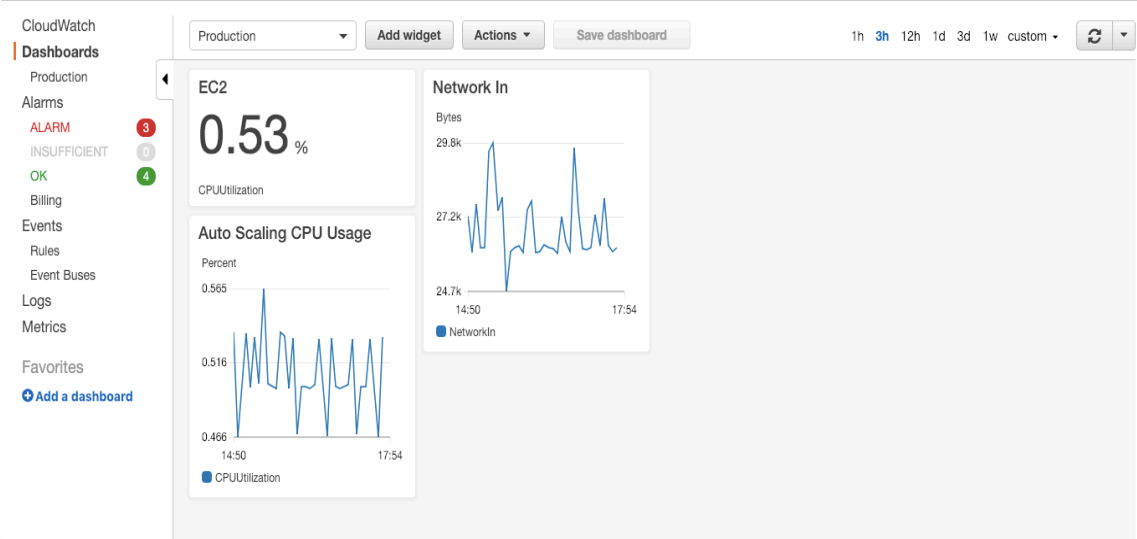


Figure 3. CloudWatch Dashboard

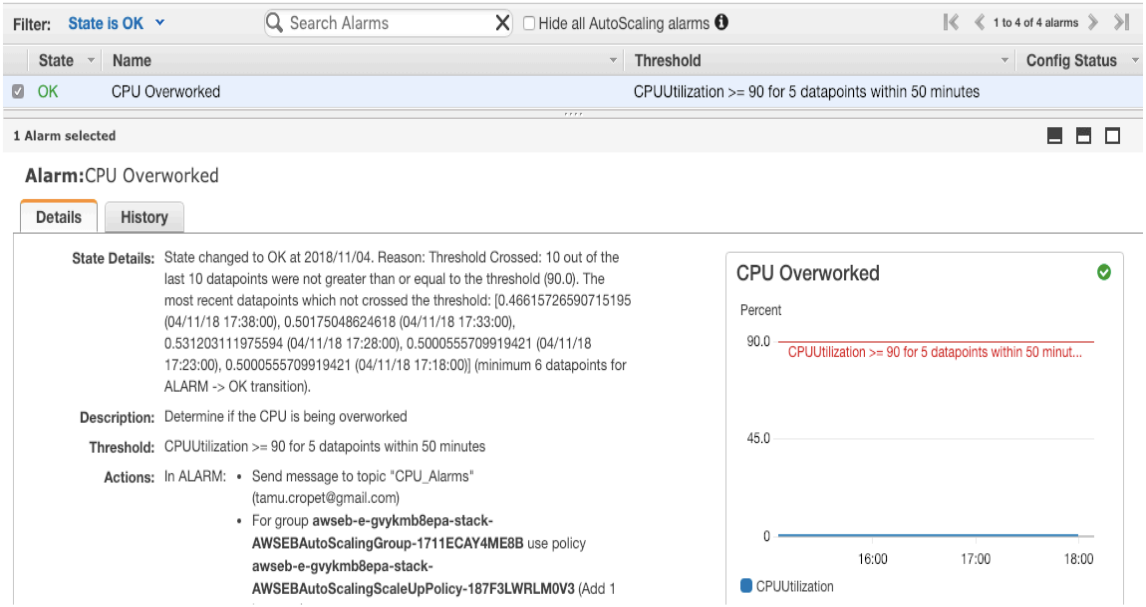


Figure 4. CloudWatch CPU Alarm

2.3.2 Configuring AWS for Deployment

There are a number of customizations necessary for the irrigation platform to run smoothly in an Elastic Beanstalk environment. The first was to set up an RDS instance outside the Elastic Beanstalk container and link it using environment variables (**Fig. 5**) and security groups. This ensures two things; first that the database is not tied to the lifecycle of an Elastic Beanstalk environment, and second, that there is only one database, which is capable of connecting to any number of Elastic Beanstalk environments. This is extremely useful for performing seamless updates with blue-green deployments. This is entirely necessary in a production environment as it ensures the site will never be down due to an update.

```
production:
  <<: *default
  adapter: postgresql
  encoding: unicode
  database: <%= ENV['RDS_DATABASE_NAME'] %>
  username: <%= ENV['RDS_USERNAME'] %>
  password: <%= ENV['RDS_PASSWORD'] %>
  host: <%= ENV['RDS_HOSTNAME'] %>
  port: <%= ENV['RDS_PORT'] %>
```

Figure 5. Database.yml and Environment Variables

The second customization enabled this platform to utilize low cost AWS services such as micro EC2 instances, which are in the free tier. Applications of this caliber have rare, yet necessary memory intensive tasks such as deployment (the building of the app). Since the app does not require a lot of memory outside of these rare instances, it is less than ideal to increase the instance size to where it is overprovisioned 99% of the time. A better solution was to add a swap space to the machine, which is easily achievable on Linux environments during runtime. This is done through three simple commands in the

command line interface (**Fig. 6**). If the application was run on one EC2 instance, one could SSH into the instance and run these commands prior to deploying the application. Obviously, this doesn't suit an environment where EC2 instances are added and removed dynamically in order to scale. Fortunately, there is a way to configure an Elastic Beanstalk environment to run commands such as these whenever an EC2 instance is created. This is done by creating an .ebextensions folder in the root of the application tree and configuration file within it. The configuration file checks to see if a swap space exists, and if it doesn't, creates it.

```
files:
  "/home/ec2-user/setup_swap.sh":
    mode: "000755"
    owner: root
    group: root
    content: |
      #!/bin/bash
      # based on http://steinn.org/post/elasticbeanstalk-swap/

      SWAPFILE=/var/swapfile
      SWAP_MEGABYTES=2048

      if [ -f $SWAPFILE ]; then
        echo "Swapfile $SWAPFILE found, assuming already setup"
        exit;
      fi

      /bin/dd if=/dev/zero of=$SWAPFILE bs=1M count=$SWAP_MEGABYTES
      /bin/chmod 600 $SWAPFILE
      /sbin/mkswap $SWAPFILE
      /sbin/swapon $SWAPFILE

commands:
  01setup_swap:
    command: "bash setup_swap.sh"
    cwd: "/home/ec2-user/"
```

Figure 6. Setup_swap.config

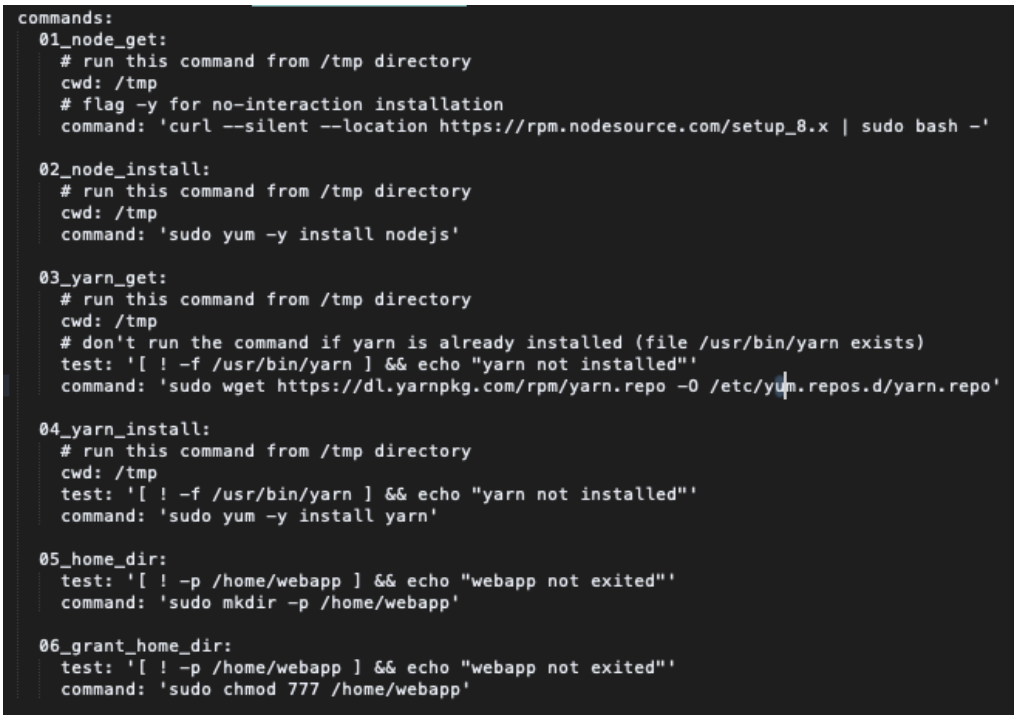
The third configuration necessary involved setting up cron tasks to handle data collection and daily reference ET calculations. Unlike the swap setup, these commands must only be executed by one EC2 instance. Otherwise there would be duplicate data

instances. To ensure this, the first EC2 instance is marked as the leader and the leader_only flag is set to true in the corresponding configuration file (**Fig. 7**). The functionality of these two rake tasks will be described in a later section.

```
files:
  "/tmp/cron":
    mode: "000644"
    owner: root
    group: root
    content: |
      0,5,10,15,20,25,30,35,40,45,50,55 * * * * /bin/bash -l -c 'cd /var/app/current && RAILS_ENV=production bundle exec rake
      fetch_data --silent'

      1 0 * * * /bin/bash -l -c 'cd /var/app/current && RAILS_ENV=production bundle exec rake calculate_et --silent'


container_commands:
  make_cron_schedule:
    command: "mv /tmp/cron /etc/cron.d/mycron"
    leader_only: true
```

Figure 7. Cron.config

The last two configuration files enable the use of Webpacker, which is a gem (library) that manages application-like JavaScript in Rails apps [6]. In this application, it allows the app to compile node modules so they can be utilized in client-side JavaScript. This was necessary for the Landsat processing method the platform utilizes. The first config file checks to see if node and yarn are installed on the EC2 instance (**Fig. 8**). If they are not, it installs them. Node.js is an open source, cross-platform JavaScript run-time environment, which executes code outside of a browser (the reason for the use of WebPacker) [7]. Yarn is a fast, reliable, and secure package manager for node that handles dependencies [8].



```
commands:
  01_node_get:
    # run this command from /tmp directory
    cwd: /tmp
    # flag -y for no-interaction installation
    command: 'curl --silent --location https://rpm.nodesource.com/setup_8.x | sudo bash -'

  02_node_install:
    # run this command from /tmp directory
    cwd: /tmp
    command: 'sudo yum -y install nodejs'

  03_yarn_get:
    # run this command from /tmp directory
    cwd: /tmp
    # don't run the command if yarn is already installed (file /usr/bin/yarn exists)
    test: '[ ! -f /usr/bin/yarn ] && echo "yarn not installed"'
    command: 'sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo'

  04_yarn_install:
    # run this command from /tmp directory
    cwd: /tmp
    test: '[ ! -f /usr/bin/yarn ] && echo "yarn not installed"'
    command: 'sudo yum -y install yarn'

  05_home_dir:
    test: '[ ! -p /home/webapp ] && echo "webapp not exited"'
    command: 'sudo mkdir -p /home/webapp'

  06_grant_home_dir:
    test: '[ ! -p /home/webapp ] && echo "webapp not exited"'
    command: 'sudo chmod 777 /home/webapp'
```

Figure 8. Yarn.config

The other configuration file needed to support Webpacker functionality involves NGINX, which is the software on the web server behind the Elastic Load Balancer (**Fig. 9**). Because Webpacker acts as a separate asset pipeline from Rails' default pipeline, NGINX must know where to look when it is asked to serve assets to clients. The rest of the settings involve setting buffer counts and sizes to avoid gateway errors.

```
files:
  "/etc/nginx/conf.d/websockets.conf" :
    content: |
      upstream backend {
        server unix:///var/run/puma/my_app.sock;
      }

      server {
        listen 80;

        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log;

        server_name production-test.us-west-2.elasticbeanstalk.com

        large_client_header_buffers 8 32k;

        location / {
          proxy_set_header X-Real-IP $remote_addr;
          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
          proxy_set_header Host $http_host;
          proxy_set_header X-NginX-Proxy true;

          proxy_buffers 8 32k;
          proxy_buffer_size 64k;

          proxy_pass http://backend;
          proxy_redirect off;

          location ^~ /assets/ {
            root /var/app/current/public;
          }

          # commented out for now in favor of rails serving the files
          location ^~ /packs/ {
            root /var/app/current/public;
          }
        }
      }

container_commands:
  01restart_nginx:
    command: "service nginx restart"
```

Figure 9. Nginx_proxy.config

2.4 Rails Application

2.4.1 Rails Configuration

Past the database configuration (telling the application to use PostgreSQL, and designating a database schema), it is necessary to create models, views, and controllers to mirror the schema. Each model file contains core rules for the model and defines relationships between other models. One can define whether or not an attribute must be

unique (i.e. a name), how attributes are ordered (by date, value etc.), dependencies, and more. Below is the model definition for Fields (**Fig. 10**). The relationships shown in figure 2 are defined accordingly. Since most of the models are similar in fashion, they are not shown here.

```
class Field < ApplicationRecord
    belongs_to :user
    validates :name, presence: true
    has_many :irrigations, dependent: :destroy
    has_many :pixel_sets, dependent: :destroy
    has_many :pixels, through: :pixel_sets
end
```

Figure 10. Field.rb (field model)

The controllers and corresponding views are much more in depth and vary greatly. The controllers house the code necessary to handle model instance creations, edits, deletions, custom functions, and pass the views any information they need. These will be covered in the application flow section as they house a good portion of the core functionality.

In order for all of the application parts to communicate properly and to work in sync, the routing of the application must be thoroughly defined. This is normally an excruciating task, but Rails 5 simplifies the process immensely. Instead of having to define all possible POST and GET requests for models, developers can simply list the model as a resource. This encapsulates all standard routes and developers are only required to add custom routes. A perfect example of Rails' convention over configuration. Some examples of standard HTTP requests are listed below along with the end state of the route configuration file (**Fig. 11, 12**).

HTTP Verb	Path	Controller#Action	Used for
GET	/fields	fields#index	display a list of all fields
GET	/fields/new	fields#new	return an HTML form for creating a new field
POST	/fields	fields#create	create a new field
GET	/fields/:id	fields#show	display a specific field
GET	/fields/:id/edit	fields#edit	return an HTML form for editing a field
PATCH/PUT	/fields/:id	fields#update	update a specific field
DELETE	/fields/:id	fields#destroy	delete a specific field

Figure 11. Standard HTTP Examples Covered by Resources Tag

```ruby
Rails.application.routes.draw do

  resources :pixels
  resources :landsats
  resources :irrigations
    resources :coefficients
    resources :crops
    resources :ets
    resources :fields
    resources :dl_readings
    resources :readings
    resources :stations
    devise_for :users
    devise_scope :user do
        get 'sign_in', to: 'devise/sessions#new'
    end

    authenticated :user do
        root to: 'welcome#index', as: :authenticated_root
    end
    root to: 'welcome#index'
    get 'overview', to: 'welcome#overview'
    get 'team', to: 'welcome#team'
    get 'irrigationlanding', to: 'welcome#irrigationlanding'
    post '/fieldirrigation', to: 'irrigations#fieldirrigation', as: 'fieldirrigation'
    post '/ndviirrigation', to: 'irrigations#ndviirrigation', as: 'ndviirrigation'
end
```

Figure 12. Route Configuration

The rest of the configuration involves the ruby gems (libraries) incorporated into the application. Most of the gems work out of the box, so to speak, but there are two that required configuring. Specifically, devise and webpacker. Below are brief explanations of each gem, how they were utilized, and what configurations were made if necessary.

- **Geocoder** - *a complete geocoding solution for ruby that features forward and reverse geocoding, IP address geocoding, access to more than 40 APIs worldwide, and basic geospatial queries [9].* This gem is used to perform reverse look ups on station coordinates to determine what county the staticon lies in.
- **Gon** - *a ruby solution for passing data from controllers to JavaScript files through the use of ajax [10].* This gem is extremely useful and utilized throughout the application. It removes the need to expose data in the views so

that it may be parsed and used within JavaScript files. Specific use cases will be covered in the following section.

- **OJ** - *Optimized JSON is a fast JSON parse and object marshal [11].* This gem is used anytime JSON is utilized in the application. Namely, when making requests to the TWDB API, and requesting large amounts of weather data from the applications database for use in visualizations such as the map.
- **Google Drive** - *a ruby library for reading and writing to files in Google Drive [12].* This gem is used to read data stored on a google sheets file, which is updated with every 15 minutes with readings from the TAMU data logger.
- **D3** – *a ruby wrapper for D3.js, which is a JavaScript library for manipulating documents based on data [13, 14]. It brings data to life through the use of HTML, SVG, and CSS.* D3 is used to dynamically create data visualizations on the platform.
- **Bootstrap** – *an open source toolkit created by Twitter for developing with HTML, CSS, and JS. It allows developers to quickly build entire apps with its SASS variables and mixins, responsive grid system, extensive prebuilt components, and powerful plugins built on jQuery [15, 16].* Bootstrap was used to extensively in the application's front-end design. It is both powerful and highly customizable making it one of the world's most popular front-end component libraries.
- **Devise** – *a flexible authentication solution based on Warden. It is a complete MVC solution designed specifically for Rails engines [17].* This gem was used to

design the user model and the authentication behind it. Devise is highly customizable and therein requires developers to specify their desired configuration when it is incorporated into an existing project. In this case, it enables registration, password confirmation, rememberable sessions, and account recovery. This is all configured in the devise initialization file. Default settings can be found on the devise homepage.

- **Webpacker** – *a library that enables the use of the JavaScript pre-processor and bundler webpack 4.x.x+ to manage application-like JavaScript in Rails [6].* This gem essentially created a second asset pipeline for the application that compiles and monitors application-like JavaScript. In this case, it is used to incorporate intricate node.js modules into the platform. The two modules utilized are Image-js and Proj4js. Their descriptions and the role they played are listed below. Most of the configuration necessary for webpacker involved AWS and was described above. Past that it was only necessary to list the required packages and ensure any file that utilized them was properly located in the new asset pipeline.
    - **Image-js** – *a full-featured image processing and manipulation library written in JavaScript [18].* This library has an amazing amount of functionality, but is simply used here to open and read specific pixel values of Landsat imagery.
    - **Proj4js** – *a JavaScript version of Proj4, which is a library to transform coordinates from one coordinate system to another [20].* This library is used to quickly transform coordinates of user fields and pixels between

the projected WGS84 coordinate system (which maps pixels from a square Landsat image to fit the curvature of the earth) and the standard WGS84 coordinate system.

2.4.2 Background Tasks

As mentioned in the AWS section, there are two background tasks the application must run constantly. One manages the collection of weather and agricultural data from stations across Texas, and the other calculates reference ET. The data collection task, fetch_data.rake, utilizes TWDB API calls to retrieve data from TWDB managed stations and accesses a continually updated Google Sheet to retrieve the latest information from Texas A&M's weather station.

The TAMU weather station consists of a Campbell Scientific CR1000X (data logger), a TE525 rain gauge, a LI-200R-SMV5 pyranometer (solar radiation), a 03002 R.M. Young wind sentry set, a HAMP60L Vaisala temperature and relative humidity probe, a RV50 Sierra Wireless Airlink 4G Industrial LTE Cellular Gateway to enable internet access, and solar panels to power it all. The CR1000x and sensors require an almost negligible amount of power to operate, but the modem pulls a full twelve volts. To minimize power consumption, the modem is only powered on for one minute every fifteen minutes. During this period, it sends an email containing current weather data. A third-party program, IFTTT, is linked to the receiving email and acts upon getting a new message from the data logger's unique email address [21]. For each message, it creates an entry in a predefined Google Sheet consisting of message's body and the time it was

received. The body of the message contains a series of key value pairs, which are separated by commas to ease parsing on the application end.

Portions of the fetch_data task relating to the TAMU station are shown below. The first creates a session with Google Drive, gathers a list of the spreadsheets within the drive, sorts them by title, and connects to the first spreadsheet in the list (**Fig. 13**). This is necessary due to the row limit Google Sheets imposes (2000 rows). Once the limit is reached, IFTTT automatically creates another sheet with the same name and an incremented number (Weather Data, Weather Data 1, Weather Data 2, etc.). When the spreadsheets are sorted by title, the newest version of the sheet is always at the beginning of the list. With the correct spreadsheet selected, the program reads the value of second column of the last row, which contains the latest email's message body. It then removes any extraneous characters such as '\n' and splits the string into substrings based on ',' locations. Variables are instantiated in case the data logger failed to report a value or reported the "not a number" error and the correct station instance is loaded into a local variable.

```
session = GoogleDrive::Session.from_config("config.json")
wsu = session.spreadsheets
wss = wsu.sort_by { |k| k.title }
ws = wss.last.worksheets[0]

station = Station.where(name: 'Texas A&M Research Farm').first
body = ws[ws.num_rows, 2]
body_readings = body.split[0]
readings = body_readings.split(',')

# Instantiate variables
batt_volt = 0
airTemp_Avg = 0
rh = 0
ea = 0
es = 0
vpd = 0
slope = 0
rswm2 = 0
rs_mj = 0
par_micromolesm2s1_Avg = 0
par_micromolesm2s1_Total = 0
rain_inches = 0
rain_mm = 0
ws_ms = 0
windDir = 0
windDir_SD = 0
logger.info 'Reading Google Drive File...'
```

Figure 13. Fetch_data.rake (1)

The next portion of the code iterates through each element in the new array and splits the key value pairs, which are separated by a colon (**Fig. 14**). A switch statement checks to see what the key is and assigns the value to a variable accordingly. The final portion checks to see if the read date is different from the latest one saved to the database, and if so, saves it (**Fig. 15**). All lines containing logger.info simply save messages to a log file.

```
readings.each do |reading|
    variable = reading.split(":")

    case variable[0]
    when "batt_volt"
        batt_volt = variable[1].to_f
    when "AirTemp_Avg"
        airTemp_Avg = variable[1].to_f
    when "AirTemp_Max"
        airTemp_Max = variable[1].to_f
    when "AirTemp_Min"
        airTemp_Min = variable[1].to_f
    when "RH"
        rh = variable[1].to_f
    when "Ea"
        ea = variable[1].to_f
    when "Es"
        es = variable[1].to_f
    when "VPD"
        vpd = variable[1].to_f
    when "Slope"
        slope = variable[1].to_f
    when "RsWm2"
        rswm2 = variable[1].to_f
    when "Rs_MJ"
        rs_mj = variable[1].to_f
    when "PAR_micromolesm2s1_Avg"
        par_micromolesm2s1_Avg = variable[1].to_f
    when "PAR_micromolesm2s1_Total"
        par_micromolesm2s1_Total = variable[1].to_f
    when "Rain_inches"
        rain_inches = variable[1].to_f
    when "Rain_mm"
        rain_mm = variable[1].to_f
    when "WS_ms"
        ws_ms = variable[1].to_f
    when "WindDir"
        windDir = variable[1].to_f
    when "WindDir_SD"
        windDir_SD = variable[1].to_f
    end
end
```

Figure 14. Fetch_data.rake (2)

```
last_reading = station.dl_readings.last
if last_reading.recordedTime !=  ws[ws.num_rows, 1]
    station.dl_readings.create(station_id: station.id, airTemp: airTemp_Avg, humidity: rh, solarRadiation: rswm2,
        rainInch: rain_inches, rainMm: rain_mm, windSpeed: ws_ms, windDir: windDir, windDirSD: windDir_SD, ea: ea,
        es: es, vpd: vpd, slope: slope, rsmj: rs_mj, parAvg: par_micromolesm2s1_Avg, parTotal: par_micromolesm2s1_Total,
        recordedTime: ws[ws.num_rows, 1], batteryVoltage: batt_volt)
    logger.info ws[ws.num_rows, 1]
end

logger.info "================== Data read complete! =================="
logger.info "\n\n\n"
```

Figure 15. Fetch_data.rake (3)

The portions of the fetch_data task which handle TWDB weather station readings are quite different. This is namely due to the fact that it is handling multiple station readings at once. The first portion defines the URL strings needed to access the TWDB API, sends a request for all current weather data, and converts the response from text to JSON format to simplify parsing (**Fig. 16**).

```
url = 'https://www.texmesonet.org/api/CurrentData'
url_humid_begin = 'https://www.texmesonet.org/api/HumidityById/'
url_humid_end = '/60'

uri = URI(url)
response = Net::HTTP.get(uri)
json = Oj.load(response)
logger = Logging.logger['data_logger']
logger.level = :info
log_name = 'fetch_data_' + Date.today.to_s + '.log'

logger.add_appenders \
    Logging.appenders.file(log_name)
```

Figure 16. Fetch_data.rake (4)

The next portion iterates through each JSON object (a station reading) and stores values accordingly (**Fig. 17**). Unfortunately, there is a bug with the TWDB API. Some of the stations do not return values for humidity despite the fact that the values are being recorded by the station. In order to retrieve these values, it is necessary to make another call to the API. This call requests humidity values by station ID. Four stations do not support humidity lookups and are skipped in this scenario. The response is again converted to JSON and the most recent value is saved. The API is also inconsistent in what it reports for sensor readings that a station lacks. Some return nil while others return blank values. There are if else statements in place to account for this. The final

portion checks to see if the read date is newer than the last reading, and if so, saves it to the database (**Fig. 18**).

```
json.each do |read|
    if read['airTemp'] == nil or read['airTemp'] == ""
        airTemp = 0
    else
        airTemp = read['airTemp']
    end

    if read['humidity'] == nil or read['humidity'] == ""
        if read['stationId'] != 10 and read['stationId'] != 11 and read['stationId'] != 12 and read['stationId'] != 13
            logger.info read['name'] +  " Required a Humidity Lookup...."
            h_url = url_humid_begin + read['stationId'].to_s + url_humid_end
            uri = URI(h_url)
            response = Net::HTTP.get(uri)
            h_json = Oj.load(response)
            humidity = h_json['values'][0]['value']
        else
            humidity = 0
        end
    else
        humidity = read['humidity']
    end

    if read['solarRadiation'] == nil or read['solarRadiation'] == ""
        solarRadiation = 0
    else
        solarRadiation = read['solarRadiation']
    end

    if read['precip'] == nil or read['precip'] == ""
        precip = 0
    else
        precip = read['precip']
    end

    if read['precip24Hr'] == nil or read['precip24Hr'] == ""
        precip24Hr = 0
    else
        precip24Hr = read['precip24Hr']
    end

    if read['windSpeed'] == nil or read['windSpeed'] == ""
        windSpeed = 0
    else
        windSpeed = read['windSpeed']
    end
```

Figure 17. Fetch_data.rake (5)

```
if read['windDirection'] == nil or read['windDirection'] == ""
    windDirection = 0
else
    windDirection = read['windDirection']
end

if read['airPressure'] == nil or read['airPressure'] == ""
    airPressure = 0
else
    airPressure = read['airPressure']
end

if read['soilMoisture'] == nil or read['soilMoisture'] == ""
    soilMoisture = 0
else
    soilMoisture = read['soilMoisture']
end

if read['soilTemperature'] == nil or read['soilTemperature'] == ""
    soilTemperature = 0
else
    soilTemperature = read['soilTemperature']
end

if read['recordedTime'] == nil or read['recordedTime'] == ""
    recordedTime = 0
else
    recordedTime = read['recordedTime']
end

if read['batteryVoltage'] == nil or read['batteryVoltage'] == ""
    batteryVoltage = 0
else
    batteryVoltage = read['batteryVoltage']
end
station = Station.where(name: read['name']).first
if station != nil
    last_reading = station.readings.last
    if last_reading.recordedTime !=  recordedTime
        logger.info station.name + " Recorded at: " + recordedTime
        station.readings.create(station_id: station.id, airTemp: airTemp, humidity: humidity,
            solarRadiation: solarRadiation, precip: precip, precipdaily: precip24Hr, windSpeed: windSpeed,
            windDir: windDirection, airPressure: airPressure, soilMoisture: soilMoisture,
            soilTemp: soilTemperature, recordedTime: recordedTime, batteryVoltage: batteryVoltage)
    end
end
```

Figure 18. Fetch_data.rake (6)

The second background task, calculate_et.rake, does exactly what the name suggests, it calculates the reference ET for all viable stations once every twenty four hours at 12:00 AM CST. As mentioned previously, the calculations used are based the Penman-Monteith Equation [32]. For the Texas A&M weather station, it is possible to use the standard version of the equation since the datalogger calculates and provides the database with both actual vapor pressure (ea) and saturation vapor pressure (es). For the

other weather stations, it is necessary to estimate ea and es using humidity values. The calculation portions of the code are shown below (**Fig. 19, 20**).

```
if station.name == "Texas A&M Research Farm"
    daily_readings = station.dl_readings.where(:created_at => time_range)
    temps = []
    humidity = []
    eas = []
    ess = []
    solarRadiationTotal = 0.0
    dailyPrecip = 0.0
    windSpeedAvg = 0.0
    airPressAvg = 0.0

    daily_readings.each do |reading|
        temps.push(reading.airTemp)
        humidity.push(reading.humidity)
        eas.push(reading.ea)
        ess.push(reading.es)

        if reading.solarRadiation < 0.0
            rs_mjm2 = 0.0
        else
            rs_mjm2 = reading.solarRadiation * 0.0009
        end
        solarRadiationTotal += rs_mjm2

        dailyPrecip += reading.rainInch
        windSpeedAvg += reading.windSpeed
    end

    windSpeedAvg = windSpeedAvg/daily_readings.length
    airPressAvg = airPressAvg/daily_readings.length
    solarRadiationAvg = solarRadiationTotal/daily_readings.length
    tempMax = temps.max + 273.15
    tempMin = temps.min + 273.15
    meanTemp = temps.inject{ |sum, el| sum + el }.to_f / temps.size
    meanHumidity = humidity.inject{ |sum, el| sum + el }.to_f / humidity.size

    julianDay = (Date.today - Date.parse("2018-01-01")).to_i + 1
    latRad = 0.0174*station.lat

    dr = 1 + 0.033*Math.cos((2*Math::PI*julianDay)/365)
    solarDeclination = (0.409*Math.sin((2*Math::PI*julianDay)/365 - 1.39))
    sunsetHourAngle = Math.acos(-Math.tan(latRad)*Math.tan(solarDeclination))

    psychrometricConstant = 0.067

    # Get average ea and es from datalogger readings
    ea = eas.inject{ |sum, el| sum + el }.to_f / eas.size
    es = ess.inject{ |sum, el| sum + el }.to_f / ess.size

    rs = solarRadiationTotal
    ra = (37.58603136*dr*(sunsetHourAngle*Math.sin(latRad)*Math.sin(solarDeclination)
        + Math.cos(latRad)*Math.cos(solarDeclination)*Math.sin(sunsetHourAngle)))
    rso = (0.75 + (0.00002*station.elevation))*ra
    slope = (4098*(0.6108*Math.exp((17.27*meanTemp)/(meanTemp + 237.3))))/((meanTemp + 237.3)**2)
    rnl = (4.903*(10**(-9)))*((tempMax**4 + tempMin**4)/2)*(0.34 - (0.14*(ea**0.5)))*(1.35*(rs/rso) - 0.35)
    rn = ((1 - 0.23)*rs) - rnl

    et = ((0.408*slope*rn) + psychrometricConstant*(900/(meanTemp+273.1))*windSpeedAvg*(es-ea)) /
        (slope + (psychrometricConstant*(1 + (0.34*windSpeedAvg))))
    et = et*0.0393701
```

Figure 19. Calculate_et.rake

```
ea = ((Math.exp((17.27*temps.min)/(temps.min + 237.3)) * (humidity.max/100))
    + (Math.exp((17.27*temps.max)/(temps.max + 237.3)) * (humidity.min/100)))/2.0
es = ((0.6108*Math.exp((17.27*temps.max)/(temps.max + 237.3)))
    + (0.6108*Math.exp((17.27*temps.min)/(temps.min + 237.3))))/2.0
```

Figure 20. Ea and Es Calculation Using Humidity

2.4.3 Application Flow

When users first access the website, they are met with the landing page (**Fig. 21**). This page presents a brief description of the tool's purpose and acts as a gateway to the rest of the site. The navbar (available on all pages) can take them to the list of weather stations (**Fig. 23**), allow them to sign in, sign out, sign up for an account, or pull up their field management page. Users are not required to sign in, but will be unable to create fields and therefore irrigation recommendations until they create an account and sign in.

Guest users are limited to the landing page, the overview page, the team page, and pages related to weather station data. The overview page and team page are pure html and hold no special functionality, merely information for the user. The weather station pages, on the other hand, are dynamically generated. There are two ways to access weather data; users can either navigate to a station's specific page, or click on a station's marker on any of the maps throughout the site. If the users wish to see historical data for the station, they must navigate to the station's page (**Fig. 22**). Each station marker has an embedded link to its page.

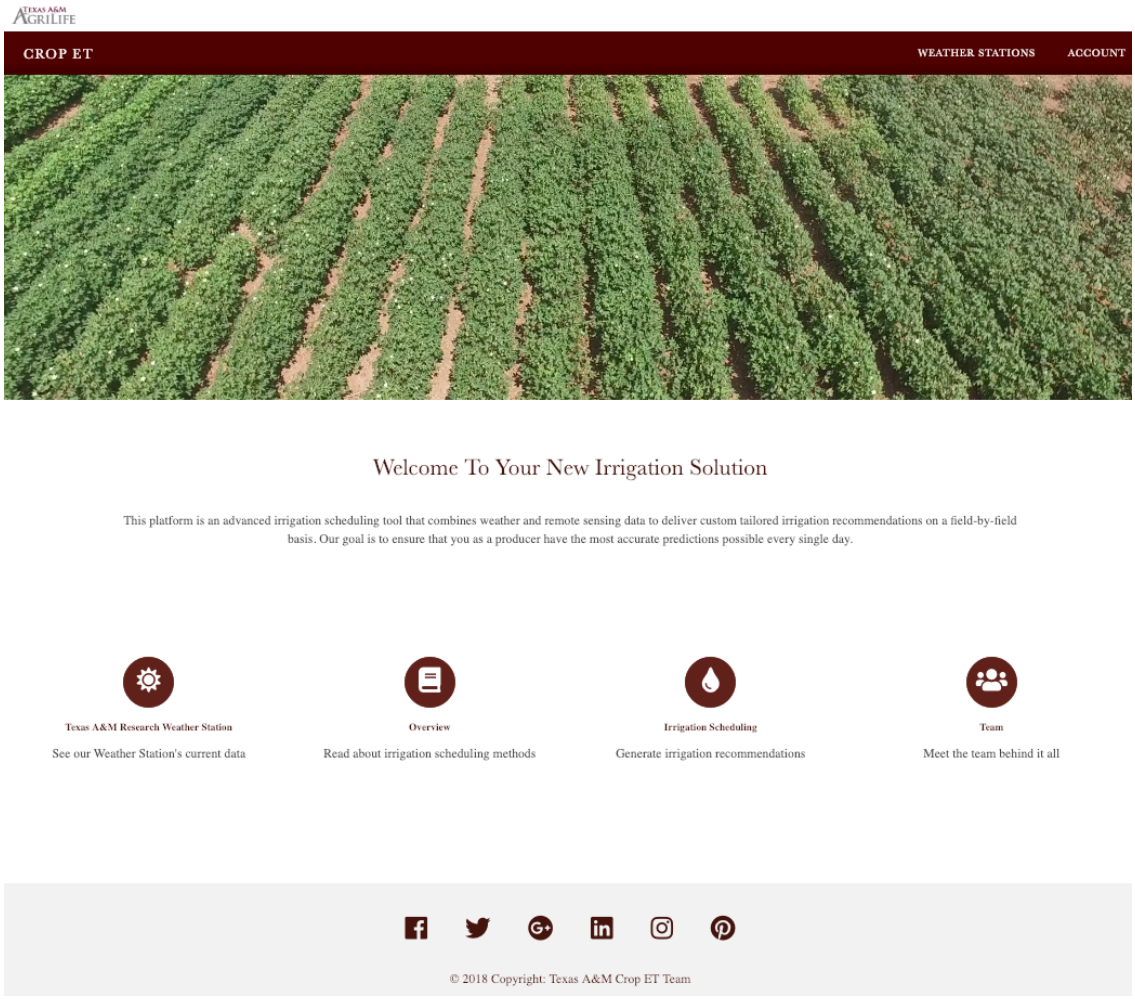


Figure 21. Landing Page

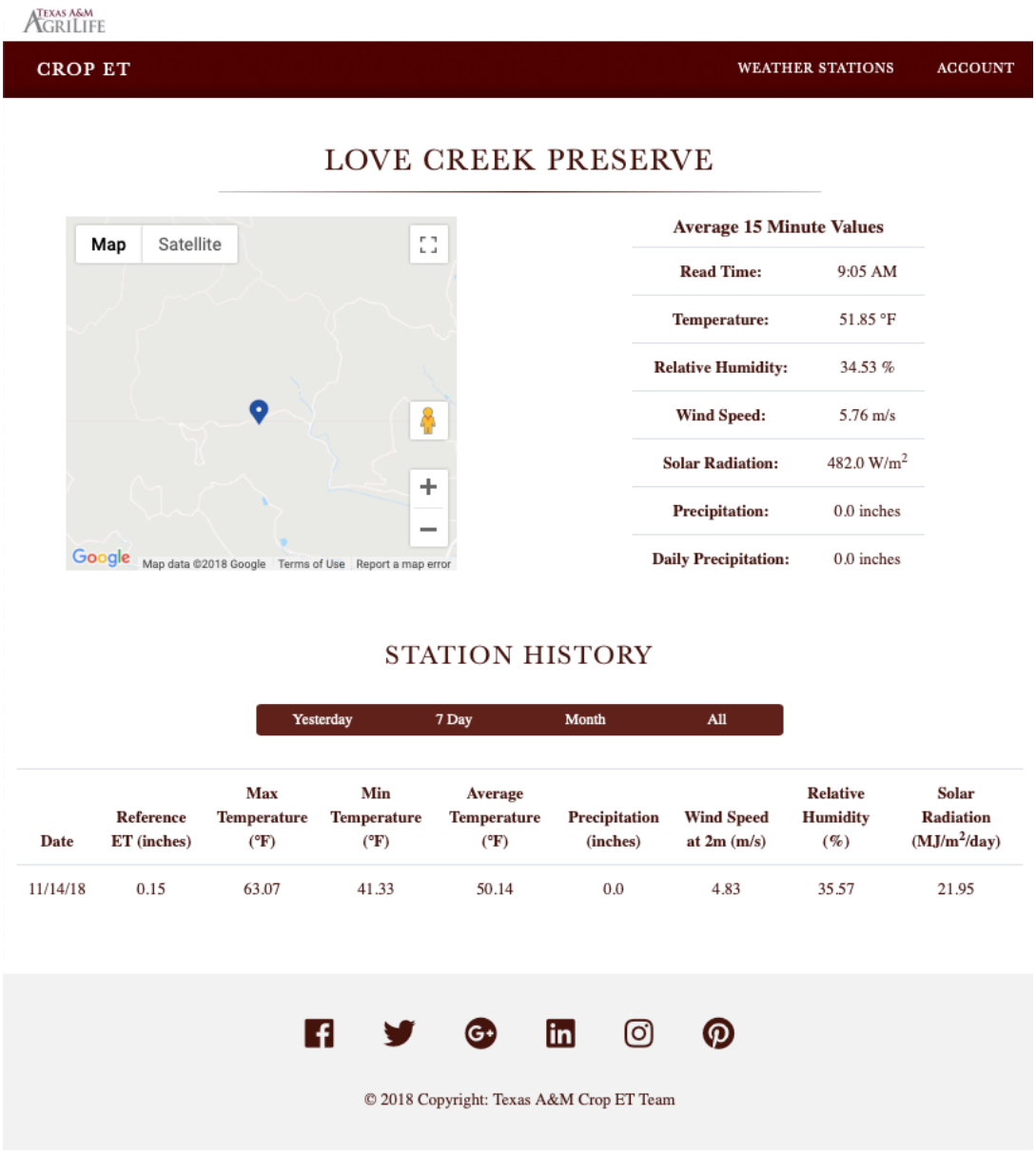
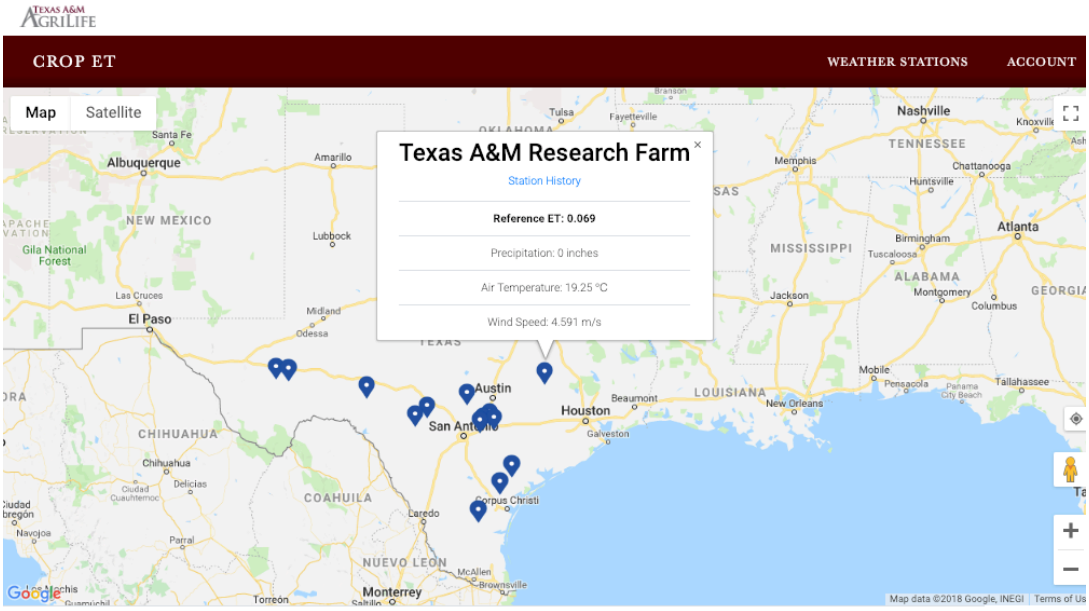


Figure 22. Station Specific Pages



County	Name	Latitude	Longitude	Elevation	
Bandera	Love Creek Preserve	29.801903	-99.417736	2269.0	Show
Blanco	Blanco Weather Station	30.08906	-98.41806	1334.0	Show
Burleson	Texas A&M Research Farm	30.551033	-96.434177	220.0	Show
Goliad	Duke Ranch	28.495129	-97.265053	101.0	Show
Guadalupe	Spring Hill Water Supply Plant	29.547216	-97.998802	514.0	Show
Guadalupe	Jones Ranch	29.534138	-97.729211	404.0	Show
Guadalupe	Strey Ranch	29.475113	-98.089391	600.0	Show
Guadalupe	Baker Ranch	29.65083	-97.83356	600.0	Show

Figure 23. Weather Stations Page

The behavior of these pages is governed largely by their associated html.erb and JavaScript files. Html.erb is a file type exclusive to Rails applications that allow you to embed ruby directly into your html code. This makes it easy to incorporate conditional behavior and dynamically build html elements like tables and forms. The html code below is responsible for the table generation on the station index page (**Fig. 24**). The variable @stations initialized in the controller and passed to the view.

```
<table class="table" style="text-align: left;">
  <thead>
    <tr>
      <th>County</th>
      <th>Name</th>
      <th>Latitude</th>
      <th>Longitude</th>
      <th>Elevation</th>
    </tr>
  </thead>

  <tbody>
    <% @stations.each do |station| %>
      <tr>
        <td><%= station.county.gsub("County", "") %></td>
        <td><%= station.name %></td>
        <td><%= station.lat %></td>
        <td><%= station.lng %></td>
        <td><%= station.elevation %></td>
        <td><%= link_to 'Show', station, class: 'btn btn-primary btn-sm', id: 'list-blue' %></td>
      </tr>
    <% end %>
  </tbody>
</table>
```

Figure 24. HTML.erb Table Generation Code

The majority of the JavaScript for the page involves standard map instantiation (examples can be found in the Google maps API documentation). I have skipped some of the html and variable declarations to save space, but the rest is shown below. The first portion checks to see if the user's browser supports geolocation (**Fig. 25**). If it does, it places a marker for the user on the map and pans the map to center on their location.

```
// ------- Geolocation Watch ---------------------------------------------

if (navigator.geolocation) {
    navigator.geolocation.watchPosition(positionSuccess, positionError, { enableHighAccuracy: true });
} else {
    $('.map').text('Your browser is out of fashion, there\'s no geolocation!');
}

function positionSuccess(position) {
    lat = position.coords.latitude;
    lng = position.coords.longitude;
    var acr = position.coords.accuracy;

    // Check to see if map needs to be updated (should only be on first update)
    if(update_map){
        map.setCenter({
            lat : lat,
            lng : lng
        });
        update_map = false;
    }

    // Add user marker to map
    var marker = new google.maps.Marker({
        position: {lat: lat, lng: lng},
        map: map,
        title: "You!",
        icon: blueCircle,
        optimized: false,
        zIndex: 999999
    })
}
```

Figure 25. Station_index.js (1)

The second portion of code iterates through all of the stations and generates the html strings needed to display current weather data in the marker's info window (**Fig. 26**). The last portion concatenates all of the strings together for the info window, creates a marker, and adds an event listener to display the info window when the marker is clicked (**Fig. 27**).

If users wish to generate irrigation recommendations, they must create an account and sign in. When they click on the links to sign in or sign up, they are met with the respective page (**Fig. 28**). The sign up page's only difference is that it contains a second box for the password confirmation. If necessary, users can request a link to reset their password. The devise gem handles these pages directly, so there is no need for a controller.

```
stations.forEach(function (d){
    weather_data = ""
    if (d.name == "Texas A&M Research Farm"){
        weather_data = '<div style="overflow: auto; height: 150px;"><table class="table"><tbody>' +
        '<tr>' + '<td style="font-weight:bold;">' + 'Reference ET: ' + tamu_et.value.toFixed(3) + '</strong></td>' + '</tr>' +
        '<tr>' + '<td>' + 'Precipitation: ' + String(dl_reading.rainInch) + ' inches</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Air Temperature: ' + String(dl_reading.airTemp) + ' ℃ </td>' + '</tr>' +
        '<tr>' + '<td>' + 'Wind Speed: ' + String(dl_reading.windSpeed) + ' m/s</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Humidity: ' + String(dl_reading.humidity) + ' %</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Solar Radiation: ' + String(dl_reading.solarRadiation) + ' MJ/m2</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Wind Direction: ' + String(dl_reading.windDir) + ' degrees</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Actual Vapour Pressure: ' + String(dl_reading.ea) + ' kPa</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Mean Saturation Vapour Pressure: ' + String(dl_reading.es) + ' kPa</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Vapour Pressure Deficit: ' + String(dl_reading.vpd) + ' kPa</td>' + '</tr>' +
        '<tr>' + '<td>' + 'RsWm2: ' + String(dl_reading.rswm2) + ' W/m2</td>' + '</tr>' +
        '<tr>' + '<td>' + 'RsMj: ' + String(dl_reading.rsmj) + ' Mj</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Battery Voltage: ' + String(dl_reading.batteryVoltage) + ' volts</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Recorded Time: ' + String(dl_reading.recordedTime) + '</td>' + '</tr>' +
        '</tbody></table><div style="overflow: auto;">';
    }
    else{
        if (d.et_valid == false){
            et_string = 'Unavailable'
        }
        else{
            et_string = latest_ets[d.id].value.toFixed(3)
        }
        weather_data = weather_data = '<div style="overflow: auto; height: 150px"><table class="table"><tbody>' +
        '<tr>' + '<td style="font-weight:bold;">' + 'Reference ET: ' + et_string + '</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Precipitation: ' + String(latest_readings[d.id].precip) + ' inches</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Air Temperature: ' + String(latest_readings[d.id].airTemp) + ' ℃</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Wind Speed: ' + String(latest_readings[d.id].windSpeed) + ' m/s</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Humidity: ' + String(latest_readings[d.id].humidity) + ' %</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Solar Radiation: ' + String(latest_readings[d.id].solarRadiation) + '</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Daily Precipitation: ' + String(latest_readings[d.id].precipdaily) + ' inches</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Wind Direction: ' + String(latest_readings[d.id].windDir) + ' degrees</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Air Pressure: ' + String(latest_readings[d.id].airPressure) + ' kPa</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Soil Moisture: ' + String(latest_readings[d.id].soilMoisture) + ' %</td>' + '</tr>' +
        '<tr>' + '<td>' + 'Soil Temperature: ' + String(latest_readings[d.id].soilTemp) + ' ℃</td>' + '</tr>' +
        '</tbody></table><div style="overflow: auto;">';
    }
```

Figure 26. Station_index.js (2)

```
var Markerinfostring = contentString1 +
        String(d.name) +
        contentString2 +
        String(d.id) +
        contentString3 +
        weather_data +
        contentString4;
var marker = new google.maps.Marker({
    position: {lat: d.lat, lng: d.lng},
    map: map,
    title: d.name,
    icon: bluePinIcon,
    id: d.id,
    content: Markerinfostring

})

google.maps.event.addListener(marker, 'click', function(){
    infowindow.setContent(this.get('content'));
    infowindow.open(map, marker);
});
```

Figure 27. Station_index.js (3)

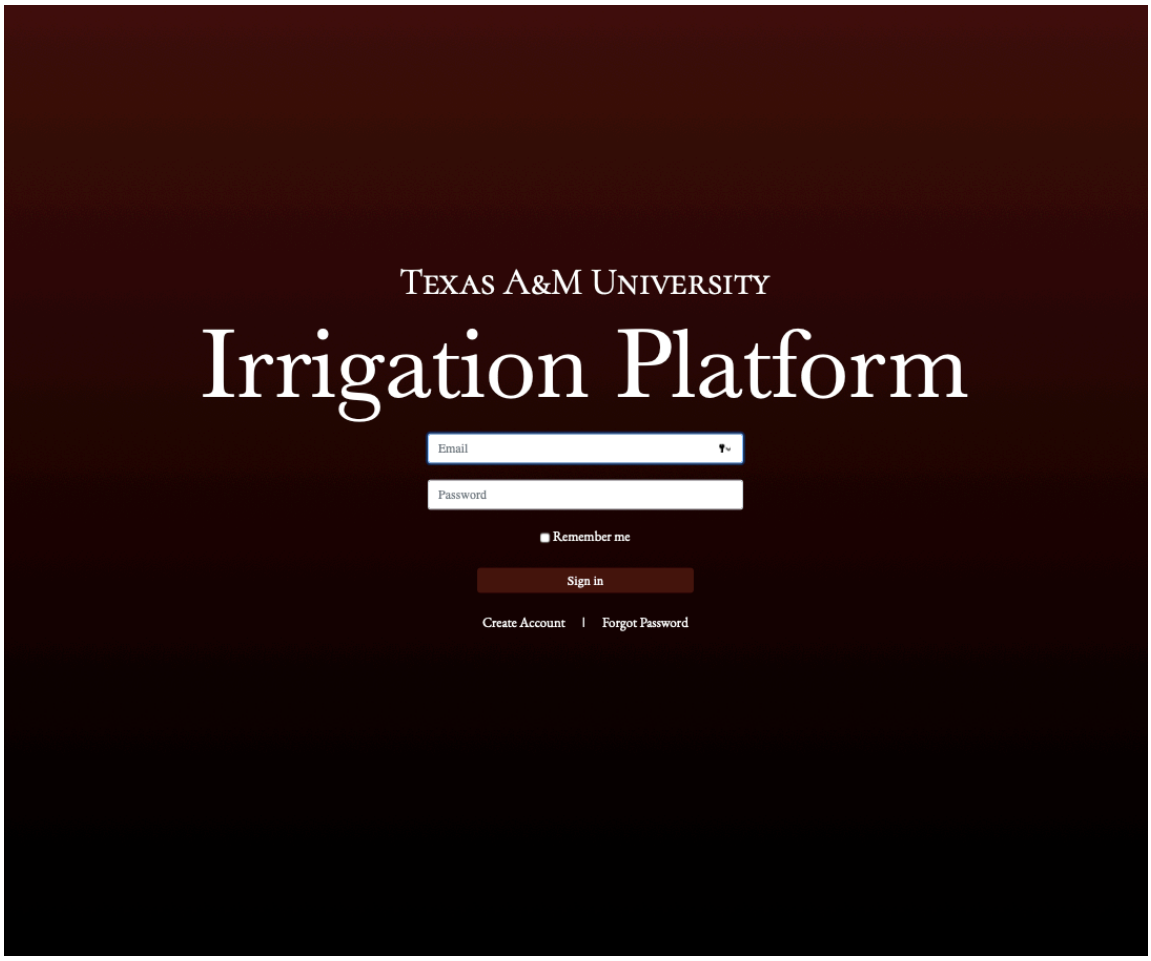


Figure 28. Sign in Page

Once users are signed in, they can create fields and generate irrigation recommendations. There are two ways to access the field management page. They can either access it directly through the navbar, or click on the "Irrigation Scheduling" button on the landing page, which will take them to the irrigation landing page (**Fig. 29**). This page contains information on how to use the tool and links for those that are adverse to using the navbar.

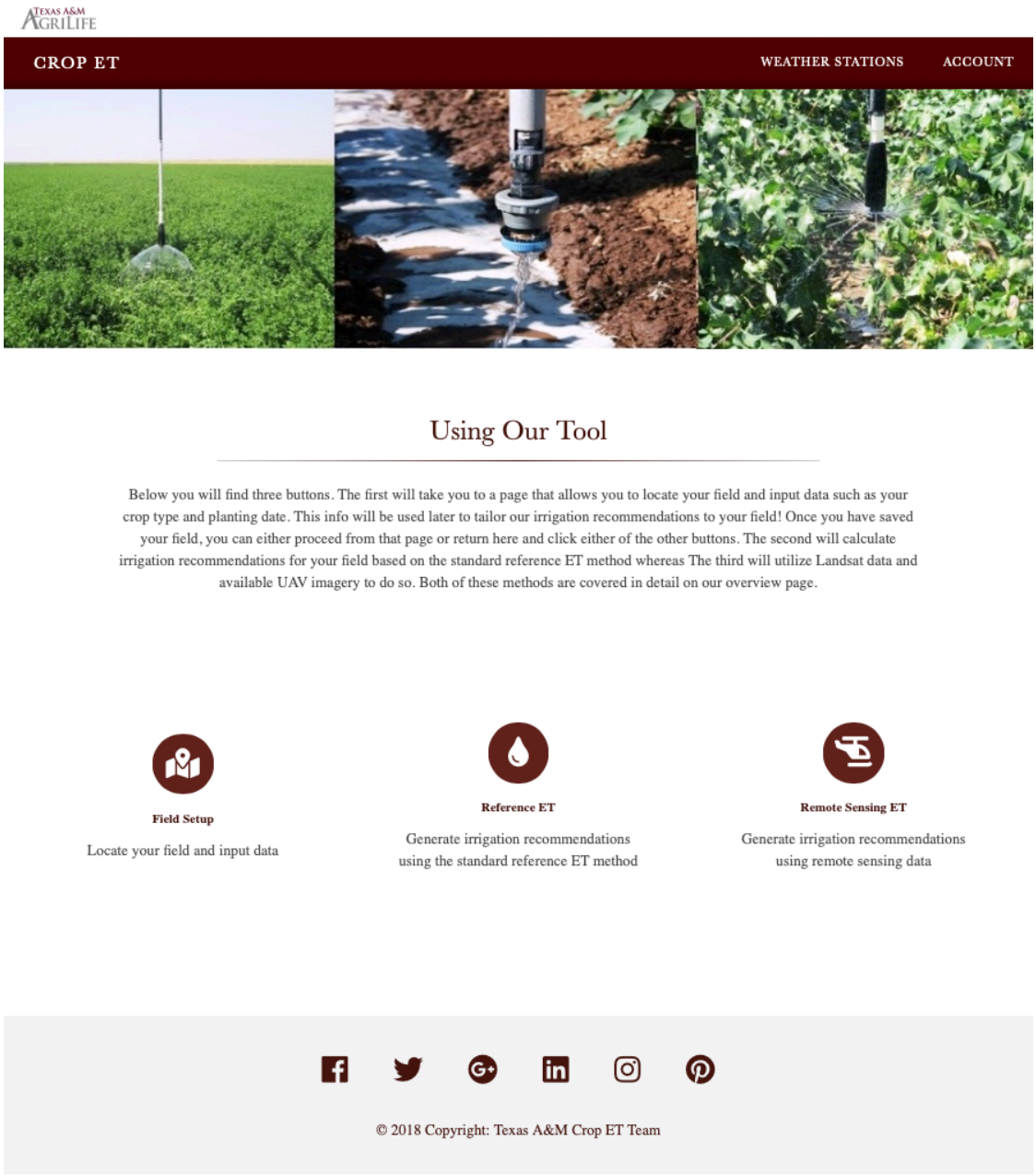


Figure 29. Irrigation Landing Page

The first step outlined is to create a field. The page for this process is shown below (**Fig. 30**). The most intuitive method is for users to navigate to their field on the google map interface. They can then select the polygon tool to create a border around their field.

Note that the polygon can be any shape and size as long as it is closed. Once the border is created, the JavaScript behind the page automatically selects the nearest station to their field. From here, the users simply need to select their crop type, planting date, and give the field a name.

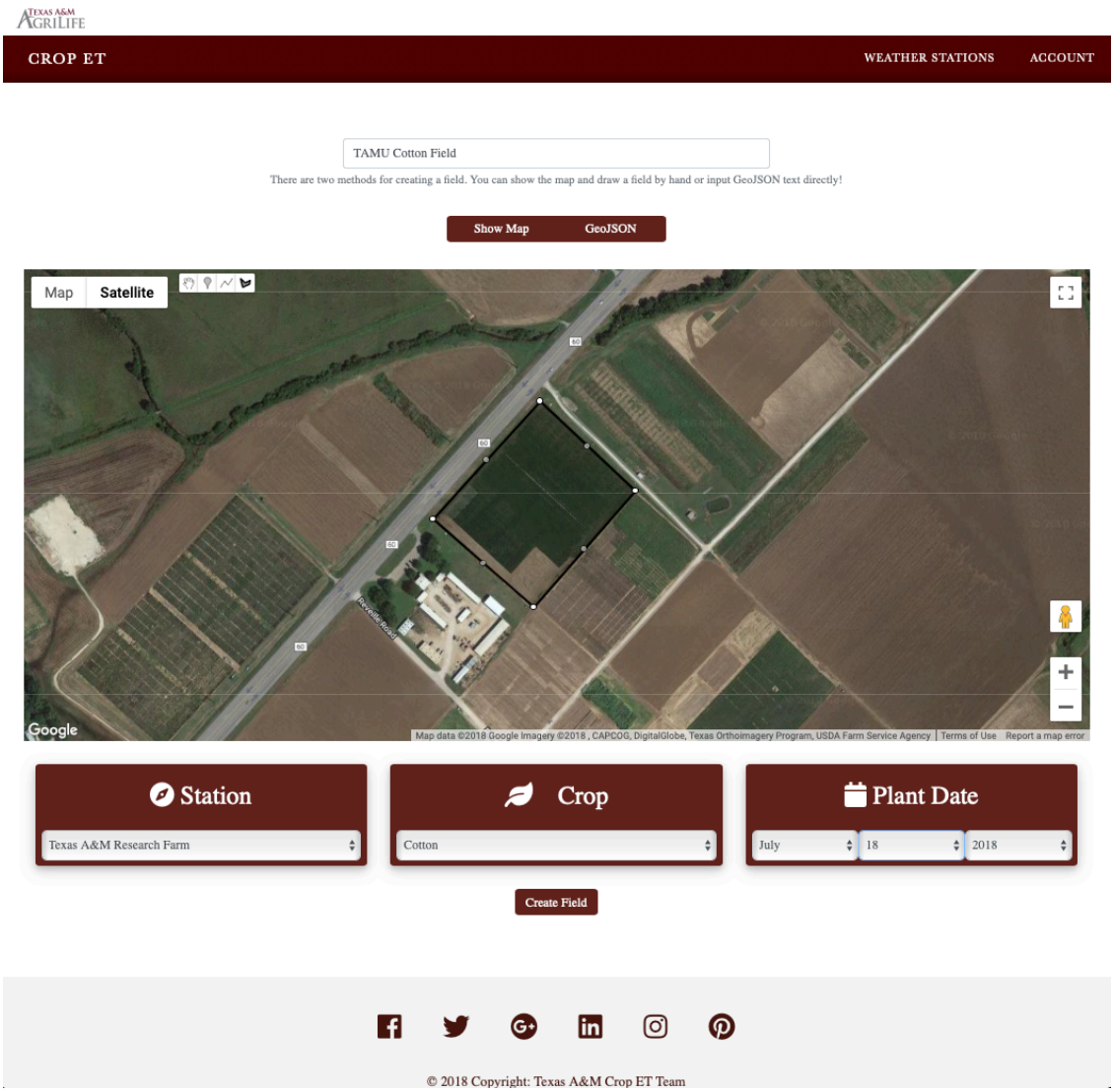


Figure 30. Field Creation Page

Nearly all of the behavior for this page is handled by its JavaScript file. Portions of this file are show below. The first portion is a function assigned to the “Show Map” button

(**Fig. 31**). It generates a google map object, sets the data controls for it, and binds listeners to the data layer so that when users are done creating the field, the code can convert the data layer's polygon object to geoJSON text, a format that can be saved to the database. The second portion of code does exactly this (**Fig. 32**). It also calculates the center point of the geometry and iterates through the stations to find the closest one. Once the closet one is found, it changes the value of the station dropdown menu on the page to match it.

```
//----- Show Map ------------------------------------------------------------
showMap.onclick = function() {

    if(first_show){
        toggledisplay("field_create_map");
        // Map instantiation
        map = new google.maps.Map(document.getElementById('field_create_map'), {
            center: {lat: 30.6097039, lng: -96.3077699},
            zoom: 12,
        });

        map.data.setControls(['Point', 'LineString', 'Polygon']);
        map.data.setStyle({
            editable: true
        });

        bindDataLayerListeners(map.data);
    }
    else{
        toggledisplay("field_create_map");
    }

}
```

Figure 31. Field_create.js (1)

```
// Save GeoJSON data
function refreshGeoJsonFromData(){
    var geoJsonInput = document.getElementById('field_geometry');

    map.data.toGeoJson(function(geoJson){
        geoJsonInput.value = JSON.stringify(geoJson, null, 2);
        var plainText = JSON.stringify(geoJson, null, 2);
        var geo = JSON.parse(plainText);
        mylat = 0;
        mylng = 0;

        for(i = 1; i < geo.features[0].geometry.coordinates[0].length; ++i){
            mylng += geo.features[0].geometry.coordinates[0][i][0];
            mylat += geo.features[0].geometry.coordinates[0][i][1];
        }
        mylng = mylng/(geo.features[0].geometry.coordinates[0].length - 1);
        mylat = mylat/(geo.features[0].geometry.coordinates[0].length - 1);

        var minDistance = 10000000000000.0;
        var closestStation = stations[0];
        stations.forEach(function (s){
            var d = distance(mylng, mylat, s.lng, s.lat);
            if(d <= minDistance){
                minDistance = d;
                closestStation = s;
            }
        });
        setSelectedValue(stationSelect, closestStation.name);
    });
}
```

Figure 32. Field_create.js (2)

The third and final portion shows the definitions for the helper functions (**Fig. 33**). The first determines the distance between two lat/long pairs by using the Haversine formula [22]. The second defines a prototype property for JavaScript's number class, which simply converts a number to its radian form. The third iterates through the station list in order to find and set the option that matches the closest station's index.

```
function distance(lon1, lat1, lon2, lat2) {
    var R = 6371; // Radius of the earth in km
    var dLat = (lat2-lat1).toRad();  // Javascript functions in radians
    var dLon = (lon2-lon1).toRad();
    var a = Math.sin(dLat/2) * Math.sin(dLat/2) +
          Math.cos(lat1.toRad()) * Math.cos(lat2.toRad()) *
          Math.sin(dLon/2) * Math.sin(dLon/2);
    var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
    var d = R * c; // Distance in km
    return d;
}

/** Converts numeric degrees to radians */
if (typeof(Number.prototype.toRad) === "undefined") {
    Number.prototype.toRad = function() {
        return this * Math.PI / 180;
    }
}

function setSelectedValue(selectObj, valueToSet) {
    for (var i = 0; i < selectObj.options.length; i++) {
        if (selectObj.options[i].text== valueToSet) {
            selectObj.options[i].selected = true;
            return;
        }
    }
}
```

Figure 33. Field_create.js (3)

Once the field is created, the user is directed to their fields page (**Fig. 34**). The map on this page shows the boundaries of all of the user's fields and places markers the closest stations. Below the map is a list of the user's fields with options to navigate to the pages for reference ET based irrigation calculations, remote sensing-based irrigation calculations, and the option to delete a field. The JavaScript that governs this page's behavior is a mixture of the functions in field_create.js and station_index.js. It iterates through the user's fields, converts the geoJSON data to a Google Maps data layer object, and adds it to the map. It also stores the center point for each polygon so once they are all placed on the map, it can zoom and pan the view to where all of the fields are visible. The marker placement is identical to what was previously shown.

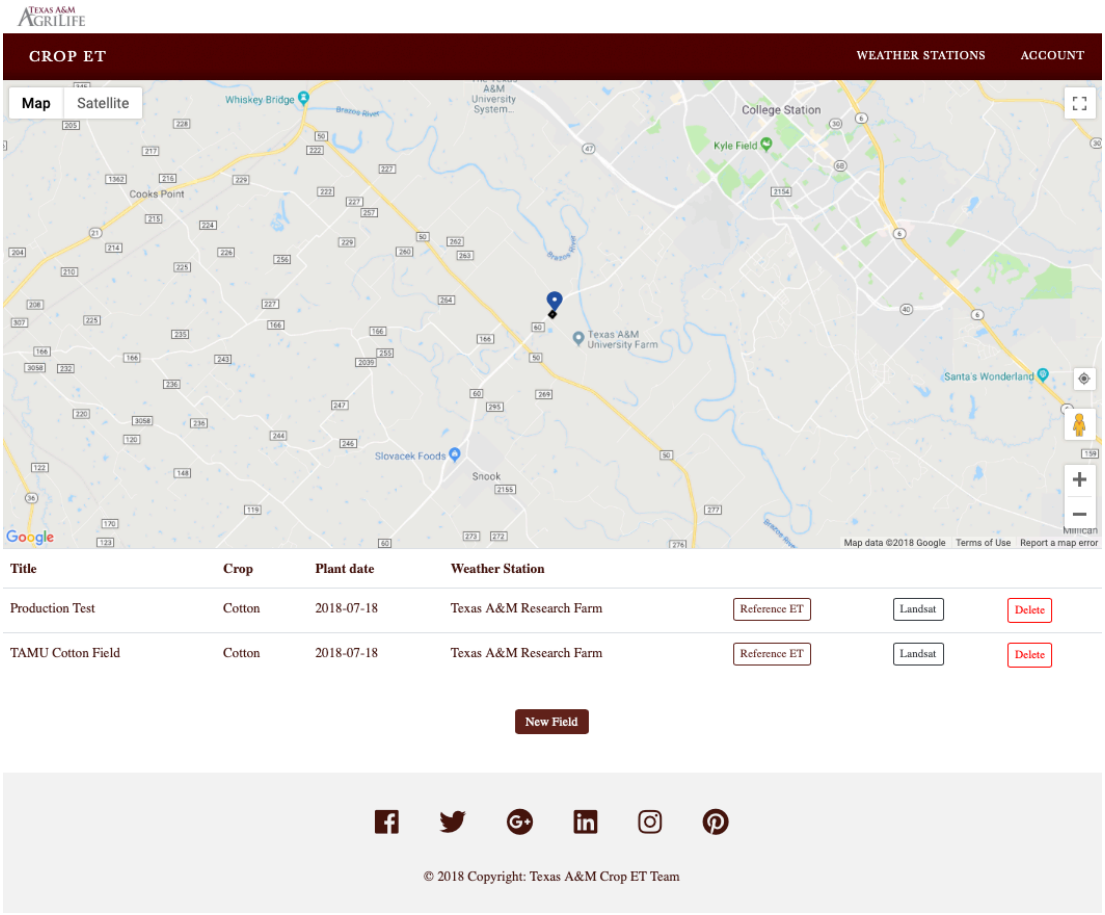


Figure 34. My Fields Page

Once a user has a field, they are able to generate irrigation recommendations. The process for generating reference ET-based recommendations is extremely simple. To do so, users simply click the "Reference ET" button that corresponds to the desired field and then click the "Generate Irrigation Recommendations" button on the following page (**Fig. 35**).

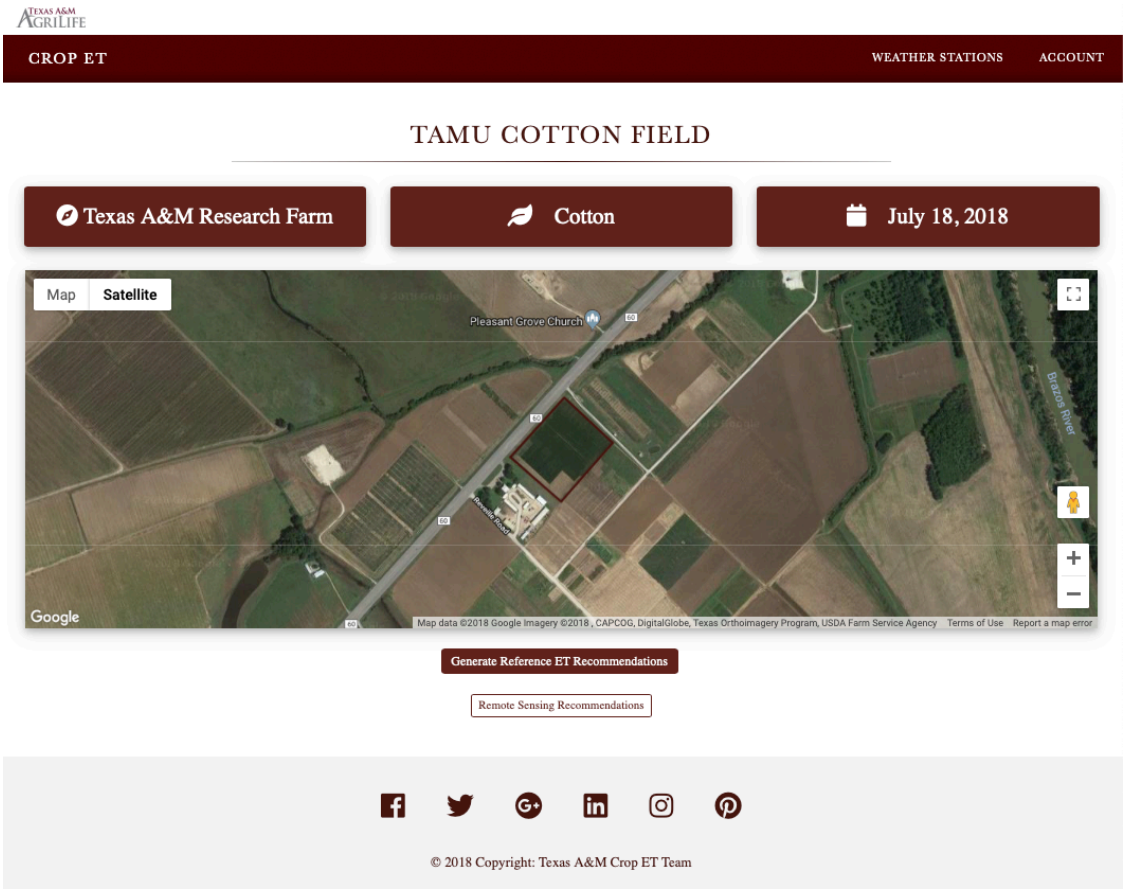


Figure 35. Reference ET Recommendation Page

Once a user clicks this button, the page sends an HTTP Post request to the server, which makes a call to the fieldirrigation function within the irrigation controller (**Fig. 36**). This call ensures that the field's id was correctly passed in the parameters of the requests and then calls the ref_et_irrigation function (**Fig. 37**).

```
def fieldirrigation
  irrigation_info = params

  if irrigation_info["field_id"] != nil
    ref_et_irrigation(irrigation_info)
  end

  @field = Field.where(:id => params[:field_id]).first

  redirect_to field_url(@field)
end
```

Figure 36. Irrigation Controller (1)

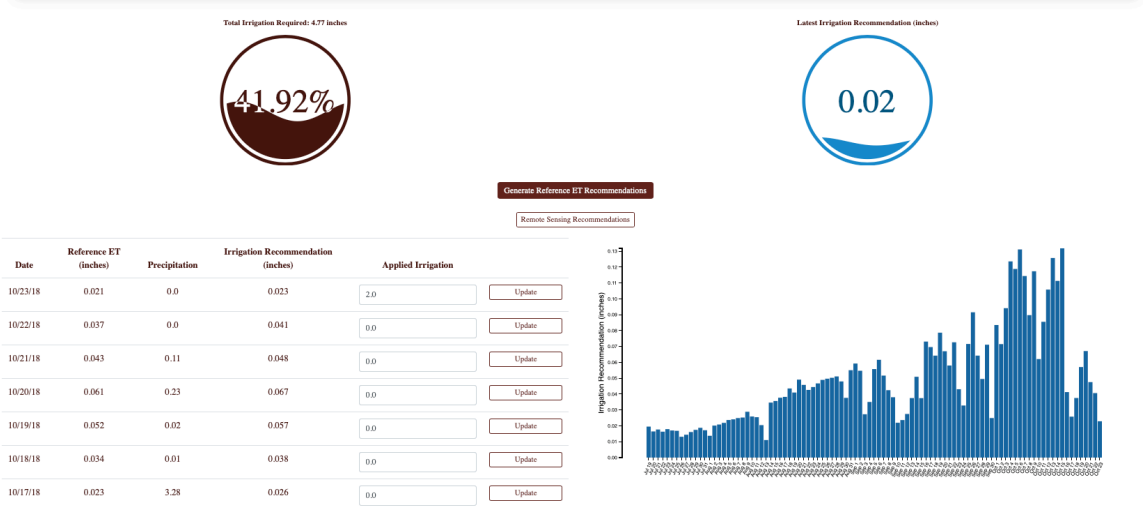
The ref_et_irrigation function performs in the following manner; first it creates a reference to the correct field using the field id given in the parameters. Next it initializes and formats all of the date variables necessary for the calculation. The code essentially starts at the current date and works its way back to the day after the crops were planting. For each day, the code checks to see if the required variables for the calculation are available (reference ET for the day and a crop coefficient corresponding to that day in the crops growth cycle). If both these variables are available, the code proceeds to the final check; to see if there has already been an irrigation calculation made for this date. If there has been, there is no need to calculate a new recommendation. This prevents duplicate values and ensures that applied irrigation values are never overwritten. If there is not an irrigation recommendation for the field on this date, it proceeds to calculate the irrigation recommendation and saves it to the database. These irrigation calculations are modeled after the previous work of Dr. Rajan [30, 31].

```ruby
def ref_et_irrigation(irrigation_info)
  # Grab and format all necessary dates for the calculation

  @field = Field.where(:id=>irrigation_info[:field_id]).first
  plant_date = @field.plant_date
  @today = Date.today
  @current_day = @today
  @days = (@today - plant_date).to_i

  # Create irrigation recommendations for every day since planting date
  while @days >= 1
    # Grab correct PET value from previous day's reading given the station.
    @et = Station.where(:id => @field.station_id).first.ets.where(created_at:
      @current_day.midnight..@current_day.end_of_day).first

    if @et == nil || (@days > 150 && @field.crop_id == 1) || (@days > 139 && @field.crop_id != 1)
      @days -= 1
      @current_day -= 1
      next
    end

    # Grab correct KC value given crop and days since planting
    @coeff = Coefficient.where(:crop_id => @field.crop_id).where(:days => @days).first
    @kc = @coeff.kc



    @irrigation = @field.irrigations.where(:remote => false).where(:created_at => @current_day.midnight).first
    if @irrigation == nil
      unless @et == nil
        # Calculate recommendation via KC*RET
        @amount_in = @kc * @et.value
        applied_irrigation = 0.0 #Always starts at 0
        @irrigation = Irrigation.new(value: @amount_in, refEt: @et.value,
          kc: @kc, precip: @et.precip, appIrr: applied_irrigation,
          created_at: @current_day.midnight, field_id: @field.id, remote: false)
        @irrigation.save!
      end
    end

    @days -= 1
    @current_day -= 1
  end
end
```

Figure 37. Irrigation Controller (2)

Once the all of the irrigation recommendations have been calculated, the function ends and returns to the fieldirrigation function where a redirect is made to cause the user's browser to reload the current page. This way the field controller can pass the newly calculated irrigation recommendations to the view. Upon reload, users are presented with a wealth of information. There are two gauges; one showing the total amount of irrigation required by the field and the percentage they have applied, and another that shows the latest irrigation recommendation (**Fig. 38**). Below the gauges is a list of all the irrigation recommendations and bar graphs that display irrigation recommendation,

reference ET, and precipitation values for each day since planting (**Fig. 39, 40**). Each entry in the table includes a spot where the user can enter the amount in inches of the irrigation they applied that day and a button to update the entry.



Date	Reference ET (inches)	Precipitation	Irrigation Recommendation (inches)	Applied Irrigation	
10/23/18	0.021	0.0	0.023	2.0	Update
10/22/18	0.037	0.0	0.041	0.0	Update
10/21/18	0.043	0.11	0.048	0.0	Update
10/20/18	0.061	0.23	0.067	0.0	Update
10/19/18	0.052	0.02	0.057	0.0	Update
10/18/18	0.034	0.01	0.038	0.0	Update
10/17/18	0.023	3.28	0.026	0.0	Update

Figure 38. Reference ET Irrigation Recommendations

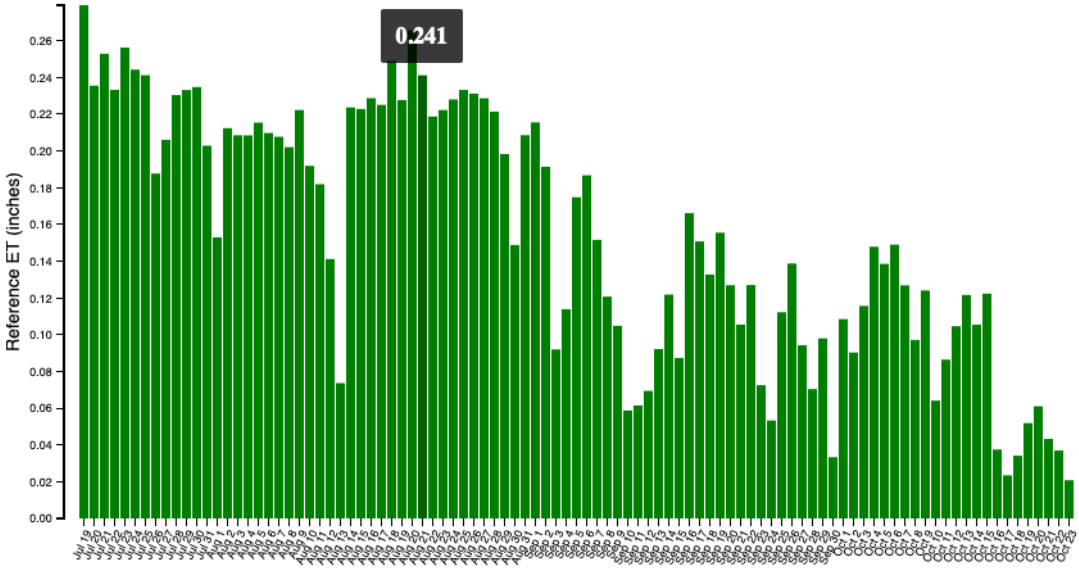


Figure 39. Reference ET Graph

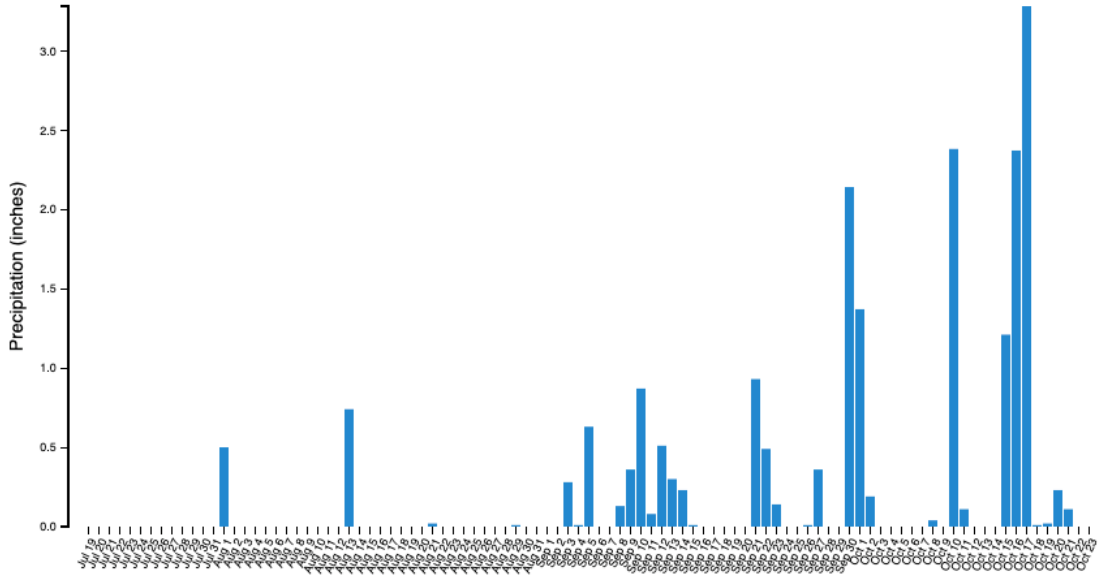


Figure 40. Precipitation Graph

The graphs and gauges are created using the d3 library and a listener function is tied to the update buttons (**Fig. 41**). Whenever a user clicks update, the function makes an AJAX call to the server so that the change is reflected on the server and then refreshes the page so the percent value in the total irrigation gauge is corrected.

```
function updateIrrigation(id){
    console.log(id);
    var text_field_name = "appIrr" + String(id);
    var field = document.getElementById(text_field_name).value;
    console.log(field);
    update_url = "/irrigations/" + String(id) + ".json"
    $.ajax({
        type : "PATCH",
        url : update_url,
        data : { irrigation: {appIrr: field} }
    })
    window.location.reload();
}
```

Figure 41. Field_show.js

CHAPTER III

REMOTE SENSING

3.1 Introduction

This chapter details how remote imagery from Landsat data is uploaded, processed and utilized to generate field specific irrigation recommendations for users. The Landsat imagery used by the platform comes from the Landsat project [23], which is a joint initiative between the U.S. Geological Survey and NASA. The project operates 8 satellites that continuously acquire "space-based moderate-resolution land remote sensing data." The newest satellite, Landsat 8, provides high quality visible and infrared images of all near-coastal landmasses on earth. It offers 16-day repetitive coverage meaning that new images can be collected and uploaded to the platform every 16 days. With these images, the platform is able to calculate a normalized difference vegetation index for every pixel. The Landsat images have a resolution of 30 meters, so the NDVI for each pixel applies to 900 m^2.

3.2 Collection and Upload

Unfortunately, the Landsat project does not provide an API so the imagery must be downloaded manually. This is done through the USGS' web-based earth explorer tool. One simply needs to create an account, search for the desired area, supply a date range, and select the Landsat 8 dataset. Users can then select the desired image set and

download the corresponding level 1 GeoTIFF data product. The data product contains images for 11 spectral bands and a text file that contains metadata for the file.

To calculate NDVI, both the red spectrum image and the near infrared image are needed. The rest of the spectral images can be discarded. In order to upload the data, Dr. Rajan or one of her graduate students must navigate to a page only accessible by them (**Fig. 42**). This page displays a form that must be filled out using data contained in the Landsat image's metadata file, and two buttons that allow the users to upload the near infrared and red images. Once the imagery is uploaded, any user can utilize it for their remote sensing-based irrigation recommendations. The form for this page was generated using Rails' built in form helper and a bit of JavaScript to show a progress bar, which tracks the progress of the image files upload to the Amazon S3 bucket (**Fig. 43**).

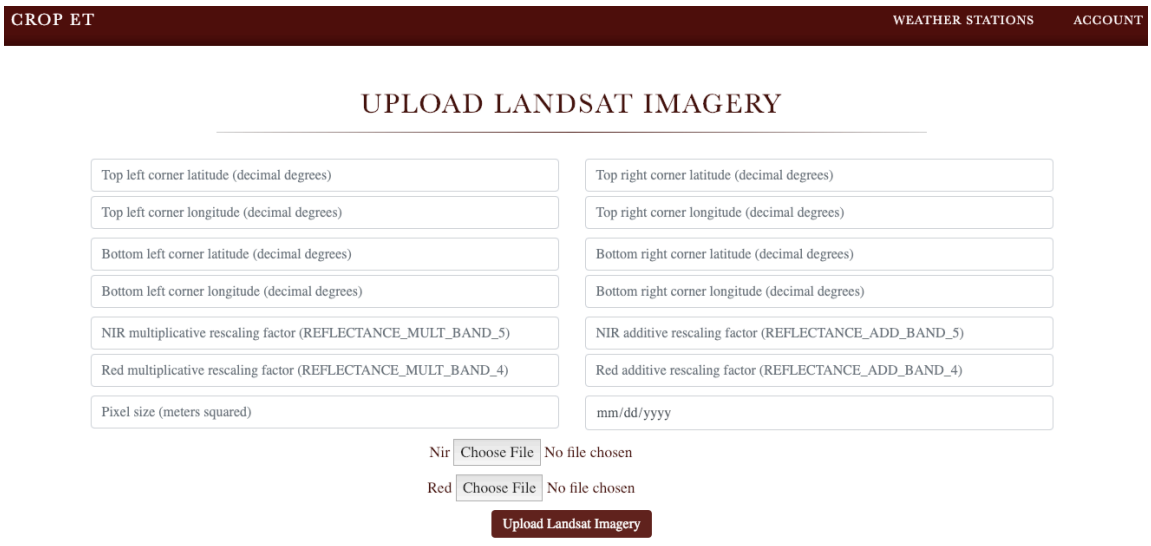


Figure 42. Landsat Upload Page

```
addEventListener("direct-upload:initialize", event => {
  const { target, detail } = event
  const { id, file } = detail
  target.insertAdjacentHTML("beforebegin", `
    <div id="direct-upload-${id}" class="direct-upload direct-upload--pending">
      <div id="direct-upload-progress-${id}" class="direct-upload__progress" style="width: 0%"></div>
      <span class="direct-upload__filename">${file.name}</span>
    </div>
  `)
})

addEventListener("direct-upload:start", event => {
  const { id } = event.detail
  const element = document.getElementById(`direct-upload-${id}`)
  element.classList.remove("direct-upload--pending")
})

addEventListener("direct-upload:progress", event => {
  const { id, progress } = event.detail
  const progressElement = document.getElementById(`direct-upload-progress-${id}`)
  progressElement.style.width = `${progress}%`
})

addEventListener("direct-upload:error", event => {
  event.preventDefault()
  const { id, error } = event.detail
  const element = document.getElementById(`direct-upload-${id}`)
  element.classList.add("direct-upload--error")
  element.setAttribute("title", error)
})

addEventListener("direct-upload:end", event => {
  const { id } = event.detail
  const element = document.getElementById(`direct-upload-${id}`)
  element.classList.add("direct-upload--complete")
})
```

Figure 43. Progress Bar JavaScript Function

3.3 Landsat Processing

The original approach to handling the Landsat involved a script that preprocessed the data and uploaded instances of the pixel model to the server. However, the sheer size of Landsat imagery made this impractical. There are roughly 60 million pixels in each spectral image. Calculating the NDVI for each pixel takes roughly one hour on a computer with a 4690K i7 intel processor, NVIDIA 1080 GPU, and 8GB of ram. While the calculation time could be cut down significantly using multithreading, it was not the bottleneck. Uploading 60 million instances of the pixel model to the database, even with

batching, took *days*. This approach also would have required the ruby controller to search the entire collection of pixels to find which ones lay within the bounds of a field. Search time would only increase as more Landsat images were processed.

Obviously, the performance of this method is unacceptable. The alternative to preprocessing the imagery is to process only what is necessary and to do it on the fly. This approach is extremely fast by comparison. Even with extremely large fields, the processing time is cut down to less than 5 minutes. The average processing time is 1-2 minutes. When users pull up the Landsat irrigation recommendation page, they are met with a list of all the Landsat images available for their field (**Fig. 44**). The list shows the date the imagery was collected, if it has or has not been processed for the user's field, and the average NDVI if it has.

When the user chooses to process one of the images, they are directed to a page that's running the necessary JavaScript script in the background. This script works in the following manner; first it loads the proj4 and image-js libraries, field information and initializes a map in a manner similar to field_show.js. Once the map is made, several variables are initialized from the Landsat model instance and field geometry. In the first portion of code shown below, the field geometry and Landsat latitude and longitude values are converted to their WGS84 projected form (x and y values) (**Fig. 45**). Then minimum and maximum x and y values are calculated in order to determine which pixels to loop through on the image.

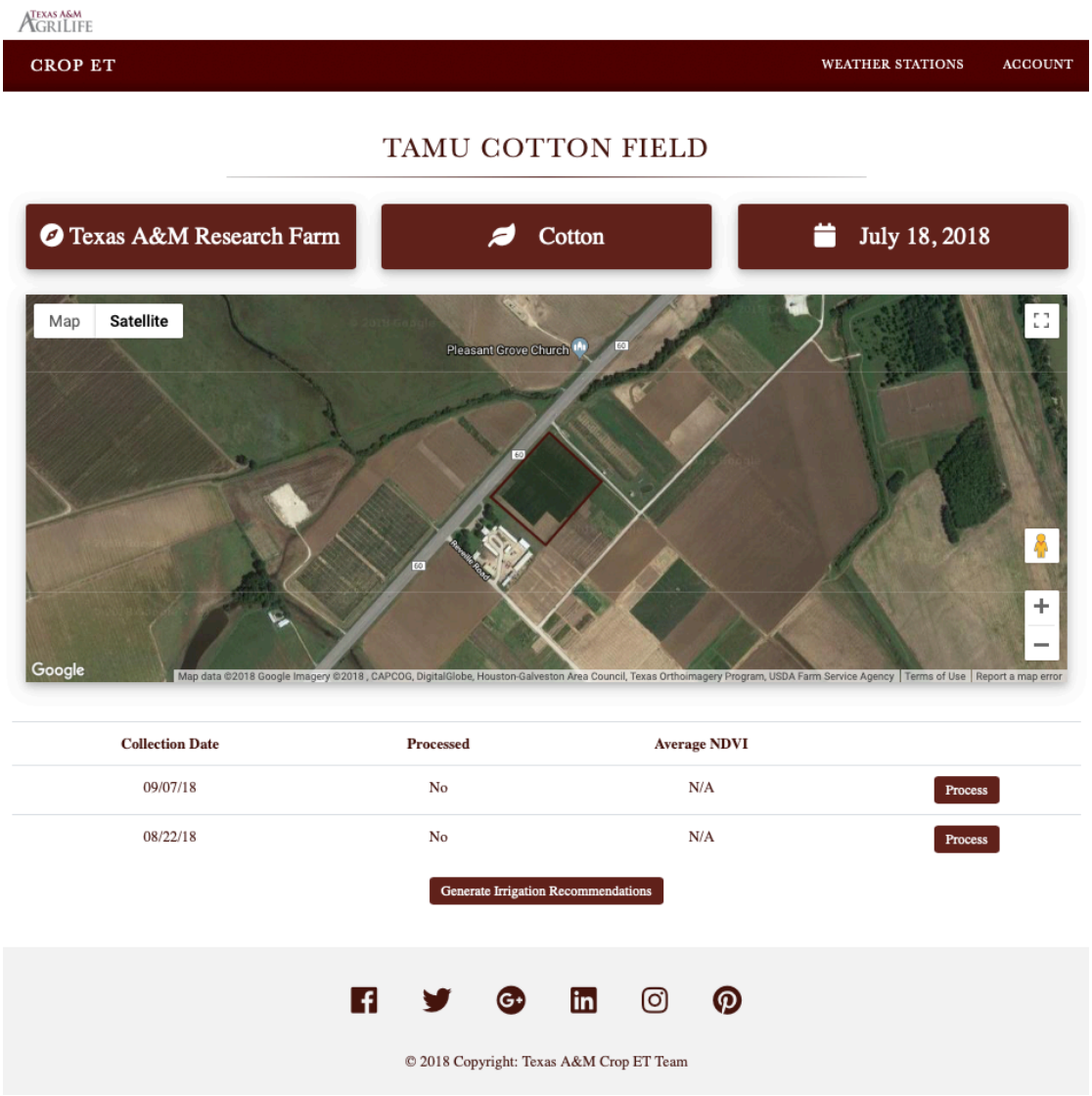


Figure 44. Landsat Irrigation Recommendation Page

```
var x_coords = []
var y_coords = []
console.log(new Date())

// Get Field Coordinates
var plainText = JSON.parse(gon.geometry)
var field_points = []
for(var i = 1; i < plainText.features[0].geometry.coordinates[0].length; ++i){
  var lng = plainText.features[0].geometry.coordinates[0][i][0]
  var lat = plainText.features[0].geometry.coordinates[0][i][1]
  coord = proj(source).forward([lng, lat])
  x_coords.push(coord[0])
  y_coords.push(coord[1])
}


move(5);

var pxmax = Math.max(...x_coords)
var pxmin = Math.min(...x_coords)
var pymax = Math.max(...y_coords)
var pymin = Math.min(...y_coords)

let nir = await Image.load(gon.nir_path)
move(10);
let red = await Image.load(gon.red_path)
move(10);


var coord = proj(source).forward([landsat.tl_lng, landsat.tl_lat])
var tl = {"x": coord[0], "y": coord[1]}
var coord = proj(source).forward([landsat.tr_lng, landsat.tr_lat])
var tr = {"x": coord[0], "y": coord[1]}
var coord = proj(source).forward([landsat.br_lng, landsat.br_lat])
var lr = {"x": coord[0], "y": coord[1]}
var coord = proj(source).forward([landsat.bl_lng, landsat.bl_lat])
var ll = {"x": coord[0], "y": coord[1]}
```

Figure 45. Process_landsat.js (1)

The other two portions show how the function loops through those pixels, and for each pixel, calculates the center point of the pixel and checks to see if that point lies within the field geometry using the Jordan curve theorem [24] (**Fig. 46, 47**). If the point is within the geometry, it grabs the pixel values from both images, converts them to top of atmosphere (TOA) reflectance [25], and calculates NDVI.

```
var pix_w = (tr["x"] - tl["x"])/(nir.width - 1)
var pix_h = (ll["y"] - tl["y"])/(nir.height - 1)
var xmin = Math.floor((pxmin - tl["x"])/pix_w)
var xmax = Math.ceil((pxmax - tl["x"])/pix_w)
var ymax = Math.ceil((pymin - tl["y"])/pix_h)
var ymin = Math.floor((pymax - tl["y"])/pix_h)
var pixels = []
var avg_ndvi = 0.0
var count = 0

var wid, len, check, r, ir, ndvi, pixel

for(var y = ymin; y <= ymax; y++){
    for(var x = xmin; x <= xmax; x++){
        if(count < 50){
          count++
          move(1);
        }
        //Find Centerpoint Lat/Lng
        wid = tl["x"] + (x*pix_w)
        len = tl["y"] + (y*pix_h)
        coord = proj(source).inverse([wid, len]);
        lng = coord[0]
        lat = coord[1]

        check = pnpoly(x_coords.length, x_coords, y_coords, wid, len);

        if(check == 1){
          //Point is within Field Geometry
          // NDVI = NIR - RED/NIR + RED
          r = red.getPixelXY(x,y)[0];
          ir = nir.getPixelXY(x,y)[0];

          //Convert to TOA Reflectance
          r = r*m_red + a_red
          ir = ir*m_nir + a_nir
          ndvi = (ir - r)/(ir + r)

          //Add up NDVI for pixel set avg
          avg_ndvi += ndvi

          //TO DO, make sure NDVI is not zero
          pixel = {"lat": lat, "lng": lng, "ndvi": ndvi};
          pixels.push(pixel)
        }
    }
```

Figure 46. Process_landsat.js (2)

```
function pnpoly(nvert, vertx, verty, x, y){
    var c = 0
    var j = nvert - 1;
    for(var i = 0; i < nvert; i++){
      if( ((verty[i] > y) != (verty[j] > y)) &&
        (x < (vertx[j] - vertx[i]) * (y - verty[i]) / (verty[j] - verty[i]) + vertx[i])){
        c = 1-c //switch between 0 and 1 whenever crossing an edge
      }
      j = i
    }
    return c // 0 means even # of edges crossed, 1 means odd.
}
```

Figure 47. Jordan Curve Theorem Function

After all NDVI values have been calculated, the function makes an ajax post to the server so that it can save all of the pixels to the database. Throughout this entire process, a progress bar is updated and displayed for the user on the Landsat processing page (**Fig. 48**).

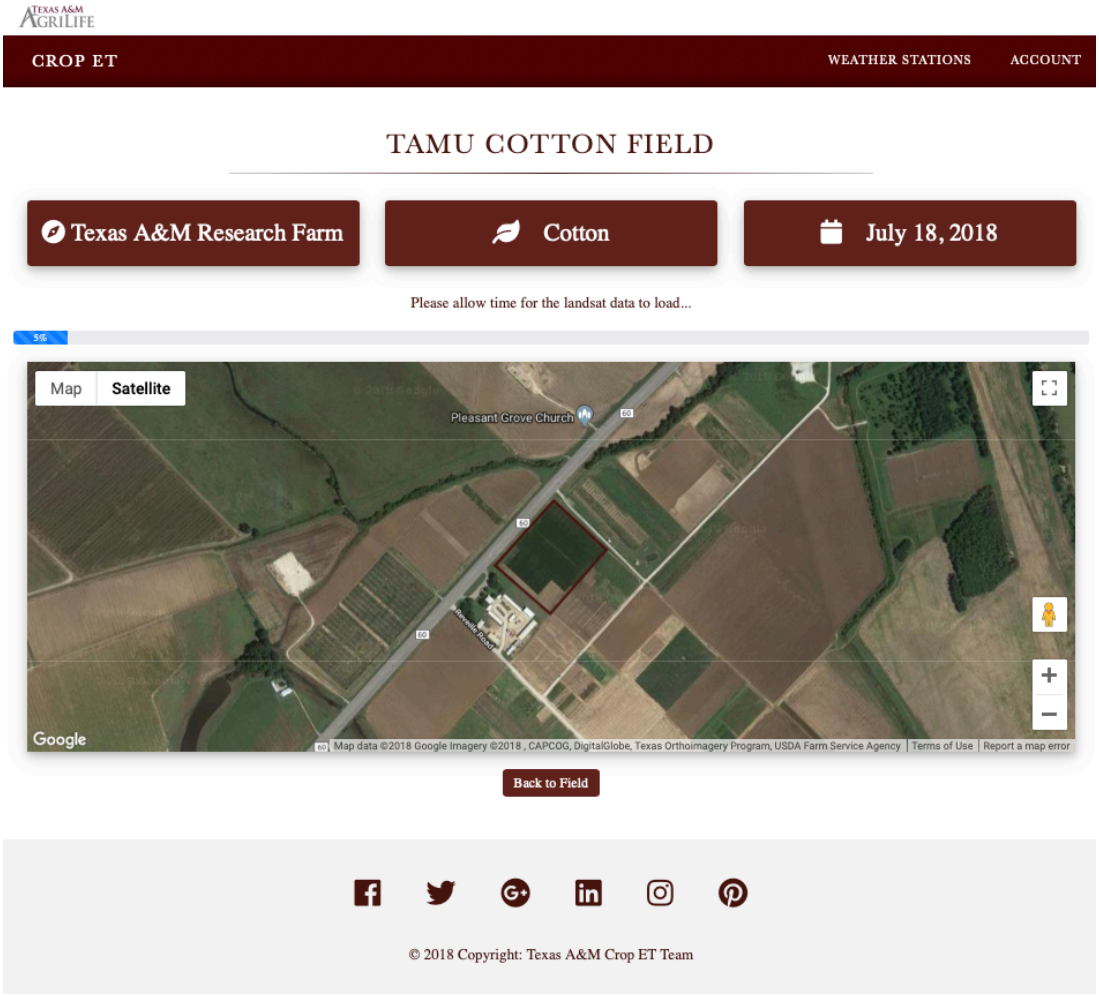


Figure 48. Landsat Processing Page

Once the pixels have been saved to the server, a function is called that creates a Google Maps rectangle for each pixel on the map and colors it on a spectrum from red to green based on its NDVI value. Each of these rectangles are assigned an event listener that

displays an info window with the corresponding NDVI value when the user hovers their mouse over them (**Fig. 49**).

Figure 49. Processed Landsat Display

Unfortunately, if Landsat images are cloudy, the user will get incorrect NDVI values. If they wish to see the images, there are download links for every single Landsat's red and near infrared images on the results page. If the user determines that their farm is obscured by clouds or simply don't like the NDVI values that were calculated, they can remove set of values from their field.

3.4 Remote Sensing-based Irrigation Recommendations

Once the user has processed at least one Landsat image for their field, they can generate remote irrigation recommendations that take the calculated NDVI values into account. The calculations utilized in the process are based on work published by Douglas J. Hunsaker and his colleagues, which outlines a linear regression relation between NDVI and the crop coefficient [26]. The controller function which performs the recommendation generation is shown below. The first portion of the function initializes

variables similar to the reference ET function (**Fig. 50**). One key difference is that this function starts the day after the planting date and works its way to the current day. It also determines the peak of the crop coefficient curve and returns its index. Last, it loops through all of the field's pixel sets and creates two arrays, which contain NDVI values and dates respectively.

```
def remote_irrigation(irrigation_info)
  # Grab and format all necessary dates for the calculation

  @field = Field.where(:id=>irrigation_info[:field_id]).first
  @plant_date = @field.plant_date
  @today = Date.today
  @current_day = @plant_date + 1.day
  @days = (@today - @plant_date).to_i

  # Grab coefficient array and find index of max
  coefficients = Coefficient.where(:crop_id => @field.crop_id).order(:days)
  coeff_max = coefficients.max_by{|x| x.kc.abs}
  coeff_max_index = coefficients.find_index(coeff_max)

  # Grab NDVI data from pixel sets
  pixel_sets = @field.pixel_sets.order(:l_date)

  ndvi_dates = []
  ndvi_values = []

  pixel_sets.each do |pset|
    day = (pset.l_date - @field.plant_date).to_i
    ndvi_dates.push(day)
    ndvi_values.push(pset.avg_ndvi)
  end
```

Figure 50. Irrigation Controller (3)

Next the function initializes the kc index, which starts at one since the loop is starting the day after planting. With everything initialized, the function can start looping through the days between the plant date and the current day. Most of the loop matches that of the reference ET function. The differences are shown in the second portion of code below

(**Fig. 51**). The function calculates the number of days since planting and checks to see if there is an NDVI value for that day. If there isn't, it just uses the existing crop coefficient value for that day. If there is, it utilizes the linear regression relation discussed earlier to calculate a new crop coefficient based on the NDVI value. Once this value is calculated the function finds the closest crop coefficient on the same side of the KC curve and updates the KC index to match its position in the array. This is essentially shifting the entire KC curve left or right so that it better aligns with the producer's crop health. Finally, the function performs the check to see if there is already an irrigation recommendation for that day, and if not, creates one and saves it to the database.

```
dsp = (@current_day - @plant_date).to_i
if ndvi_dates.include? dsp
  ndvi = ndvi_values[ndvi_dates.find_index(dsp)]
  @coeff = 1.49*ndvi - 0.12
  if @coeff > 0.0
    # Landsat data was good enough to use. Modify KC index as needed
    # See which half of the bell curve this lies on
    if @kc_index < coeff_max_index
      # First half of bell curve
      closest_kc = coefficients[0..coeff_max_index].min_by{|x| (@coeff-x.kc).abs}
      @kc_index = closest_kc.days
    else
      # Second half of the bell curve
      closest_kc = coefficients[coeff_max_index..coefficients.length].min_by{|x| (@coeff-x.kc).abs}
      @kc_index = closest_kc.days
    end
  else
    @coeff = Coefficient.where(:crop_id => @field.crop_id).where(:days => @kc_index).first.kc
  end

else
  @coeff = Coefficient.where(:crop_id => @field.crop_id).where(:days => @kc_index).first.kc
end
```

Figure 51. Irrigation Controller (4)

CHAPTER IV

RESULTS AND FINDINGS

4.1 ET and Irrigation Recommendation Calculations

To ensure the correctness of our reference ET and irrigation recommendation calculations, Dr. Rajan, her students, and I have monitored the platform for roughly three months. Reference ET calculations are checked daily and corroborated via third party services, which calculate reference ET in nearby areas. In addition to this, the platform maintains detailed logs of all reference ET calculations. Our team has randomly sampled dozens of calculations and validated the results by hand. We used similar tactics to ensure that our irrigation recommendations are generated correctly. In one case, we generated irrigation recommendations for one of Dr. Rajan's cotton fields and compared the results to her student's manual calculations. The generated recommendations were well within the target range.

4.2 NDVI Calculations

The standard method for calculating NDVI using Landsat data involves the use of software such as ArcGIS, which enables users to perform complex geospatial analysis [27]. Using ArcGIS, our team calculated NDVI for several Landsat image sets and compared the results to the platform's calculations. All of the values matched. ArcGIS also enables users to see the latitude and longitude of any pixel. We used this to ensure that the calculated latitude and longitude values for pixels on the platform were correct as well. Once again, everything checked out.

4.3 User Study

To ensure the platform met project goals, we performed a user study with both expert and naïve users. The study consisted of a cognitive walkthrough method and usability testing. The cognitive walkthrough method "combines software walkthroughs with a cognitive model of learning through exploration [23, 24]." In a cognitive walkthrough, the developer of an interface walks a user through an application in the context of core tasks a typical user would want to accomplish. The actions and respective feedback from the interface are compared to the user's goals and previous knowledge. Any discrepancies between the user's expectations and the actual outcome are documented. The project goals our user study tested are listed below:

- The application must allow users to quickly access current agriculture and weather data
- The application must allow users to locate and designate the boundaries of their field
- The application must provide users with field-specific reference ET based irrigation recommendations
- The application must provide users with field-specific remote sensing-based irrigation recommendations

The user study involved four tasks that corresponded to a specific project goal. After each task, users were asked if they were able to complete the task, how long it took them

to complete the task, if that time frame was reasonable, and if there was any feedback they wished to give regarding the process. The first task directed users to the website and asked them to look up the current weather and agriculture data for any station. All users were able to complete this task and did so in under a minute. Users also felt that the irrigation platform performed the same or better than alternatives in regards to this task. Each of them felt the time frame for this task was reasonable.

The second task asked users to find and create a field. Once again, all users were able to complete this task. However, the time to do so varied. Oddly enough, experience level in agriculture had nothing to do with the speed at which a user completed the task. Some users were simply more accustomed to highly interactive web interfaces. The results are shown below (**Fig. 52**). There are no other platforms that allow users to accomplish this task. So, it is impossible to compare the irrigation platform's performance against alternatives. However, all of the users felt that the time frame required to complete this task was reasonable.

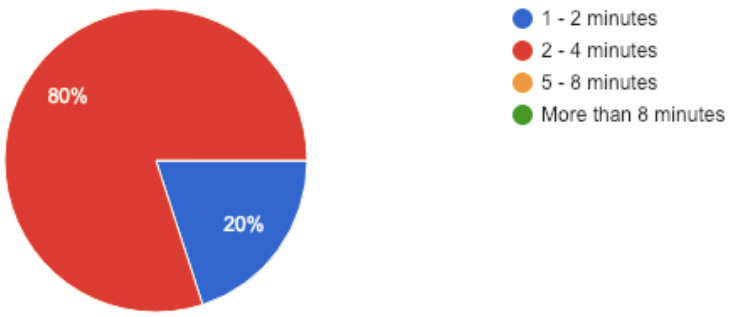


Figure 52. Task 2 Timing Results

The third task asked users to generate irrigation recommendations for the field they created in task 2. Once again, all users were able to accomplish the task. The timing results showed that all users were able to do so in under a minute. Each of the users felt that this time frame was reasonable and again there were no known alternative ways to accomplish this task. All users felt that the recommendations were presented to them in a clear and concise manner.

The final task required users to generate remote sensing-based irrigation recommendations for the same field. All users were able to process Landsat data for their field and generate the appropriate irrigation recommendations. However, the time to complete the task varied (**Fig. 53**). Naïve users tended to only process one Landsat image for the field while expert users processed as much of the imagery as possible. Once again, there are no other platforms that allow users to accomplish this task. And while all users felt the time frame for accomplishing this task was reasonable, they did not feel that the recommendations were presented clearly. The results looked too similar to the reference ET recommendations. This has been considered and steps are being taken to clarify key differences in the results.

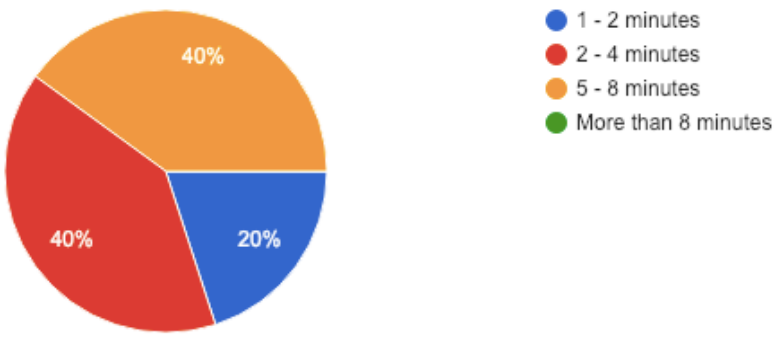
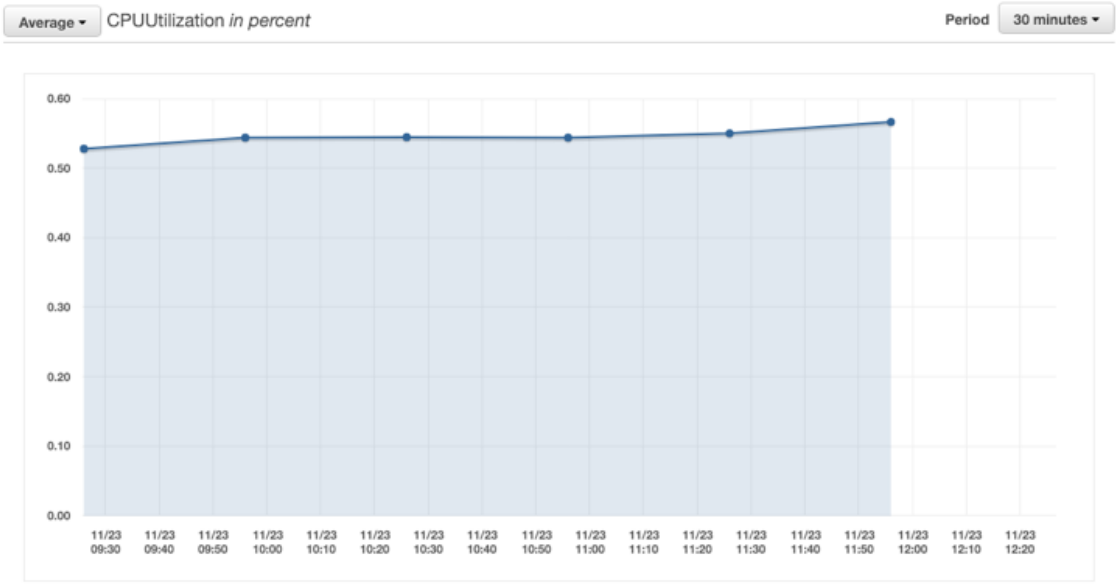


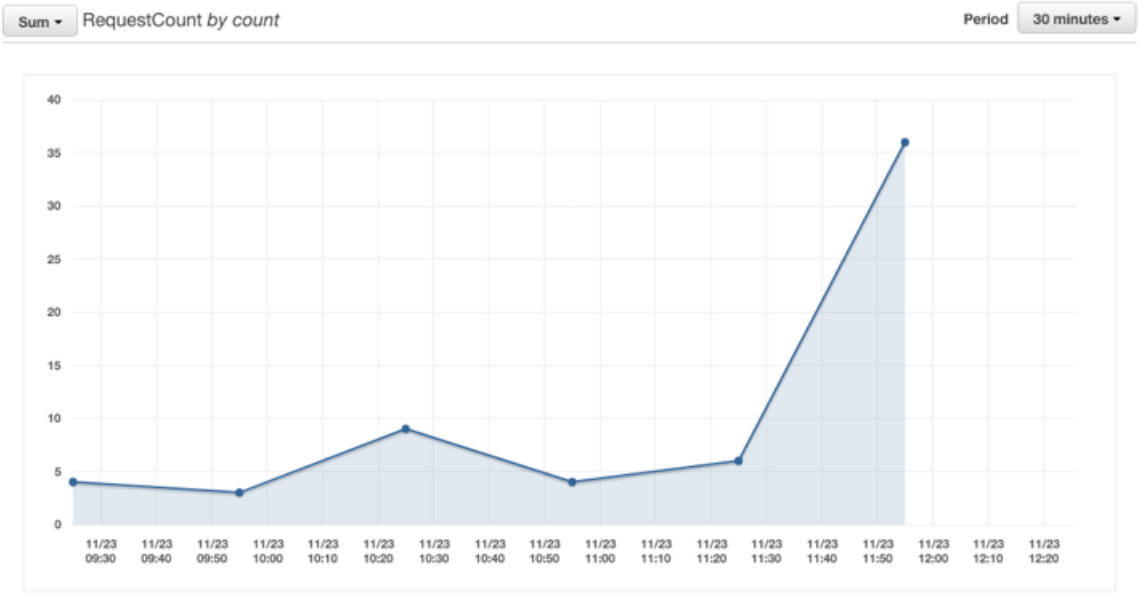
Figure 53. Task 4 Timing Results

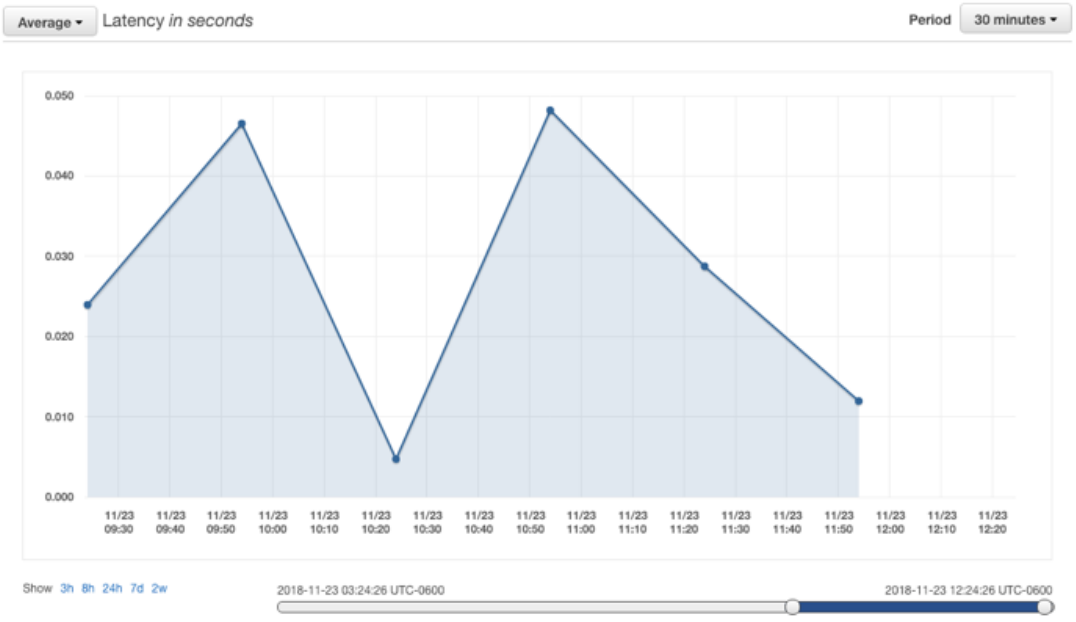
4.4 Server Performance

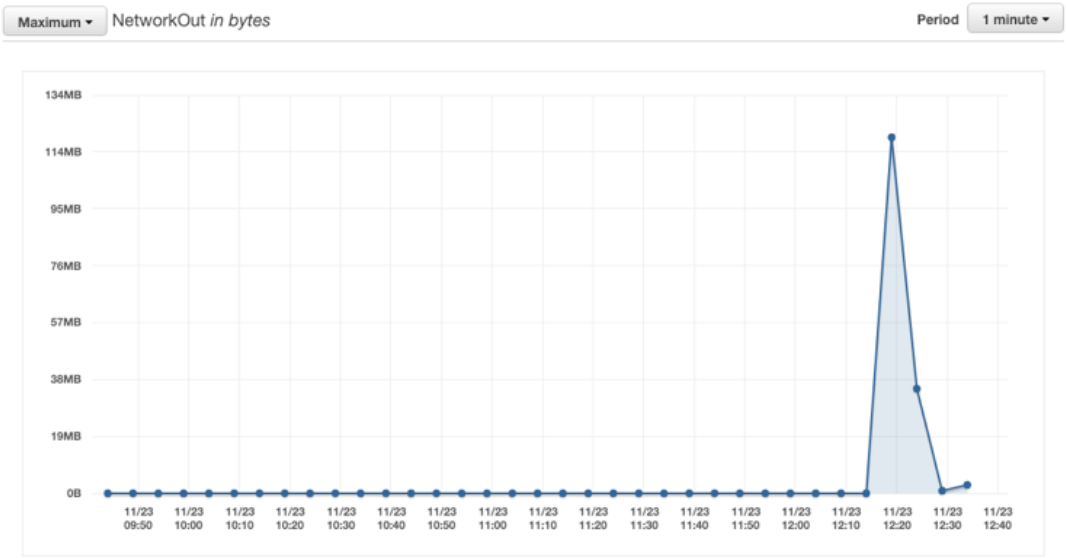
During the user study, the elastic beanstalk instance was monitored in real-time. A number of metrics were measured to see if any more auto scaling mechanisms needed to be put in place. It seems that for now, the platform can handle upwards of 40 concurrent users at a time without exhibiting stress in the slightest. As you can see below, all metrics were reasonable and well below desired thresholds (**Fig. 54**).

1.0	0.6%	19.8	42.0	1MB	119MB
Healthy Host Count	CPU Utilization	Average Latency *in Milliseconds*	Sum Requests	Max Network In	Max Network Out






Average
Latency in seconds
Period
30 minutes
0.050
0.040
0.030
0.020
0.010
0.000
11/23 09:30
11/23 09:40
11/23 09:50
11/23 10:00
11/23 10:10
11/23 10:20
11/23 10:30
11/23 10:40
11/23 10:50
11/23 11:00
11/23 11:10
11/23 11:20
11/23 11:30
11/23 11:40
11/23 11:50
11/23 12:00
11/23 12:10
11/23 12:20
Show 3h 8h 24h 7d 2w
2018-11-23 03:24:26 UTC-0600
2018-11-23 12:24:26 UTC-0600


Maximum
NetworkOut in bytes
Period
1 minute
134MB
114MB
95MB
76MB
57MB
38MB
19MB
0B
11/23 09:50
11/23 10:00
11/23 10:10
11/23 10:20
11/23 10:30
11/23 10:40
11/23 10:50
11/23 11:00
11/23 11:10
11/23 11:20
11/23 11:30
11/23 11:40
11/23 11:50
11/23 12:00
11/23 12:10
11/23 12:20
11/23 12:30
11/23 12:40

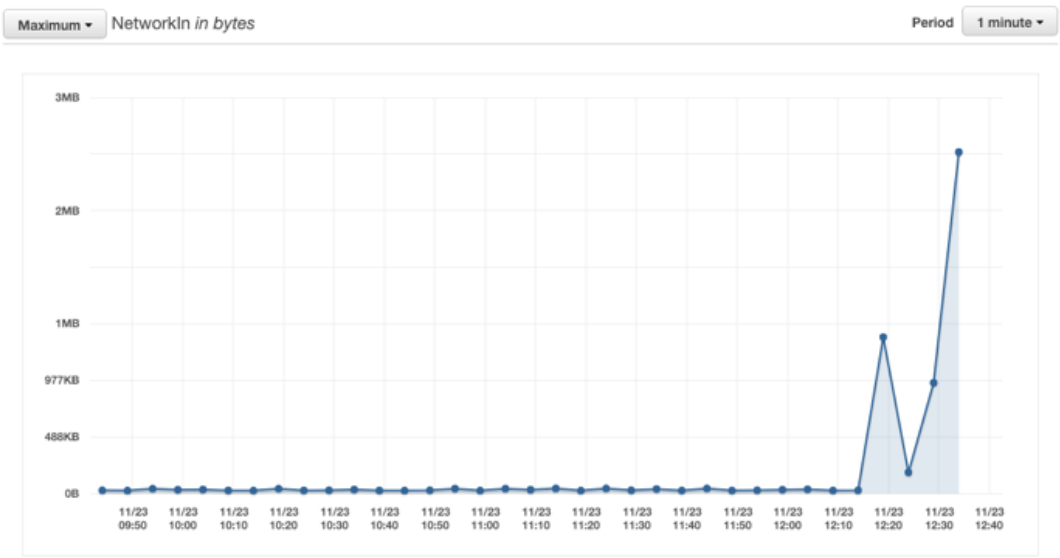


Figure 54. Elastic Beanstalk Load Test Metrics

CHAPTER V

SUMMARY AND CONCLUSION

5.1 Summary

As discussed previously, there is a need to amplify irrigation scheduling efficiency for agriculture producers in Texas. The best way to accomplish this is to utilize real-time plant and weather conditions to calculate irrigation needs on a field-by-field basis. We present a robust irrigation platform that manages this through a combination of continually updated weather station data, remote image data, and limited user interaction. The platform was developed using Ruby on Rails and modified to function on Amazon's Web Services. It automatically collects weather data from stations across Texas every 15 minutes and allows the administrators to easily upload new Landsat imagery. Through an extensive user study, the platform was shown to (a) provide users with real time weather and agricultural data, (b) allow users to easily and efficiently designate the location of their field on a Google Map interface, (c) generate and clearly present accurate reference ET-based irrigation recommendations, (d) allow users to process Landsat imagery for their field in order to determine NDVI, (e) generate and clearly present accurate remote sensing-based irrigation recommendations, and (f) allow users to keep track of their irrigation values throughout the growing season. Performance metrics and user feedback show that the application handles exceptionally well and performs task in a reasonable time frame.

5.2 Conclusion

There has been significant effort in agricultural research regarding irrigation scheduling efficiency [21, 25, 26]. Many of these efforts have produced promising results. Unfortunately, the results and irrigation scheduling methods conceived during research rarely make it into the hands of every day producers. Additionally, these methods are time intensive and require those practicing them to have access to a wealth of data, which in general, is not easily accessible. The platform presented in this research addresses both of these issues. It automatically collects current weather and agricultural data, allows easy upload of remote imagery, and procedurally generates field-specific irrigation recommendations for producers using advanced irrigation scheduling methods. The platform is efficient, intuitive, and available to use for any producer in Texas. This platform will enable producers to irrigate their fields efficiently and drastically reduce the cost of calculating irrigation needs for researchers.

REFERENCES

[1] Reenskaug, Trygve. “MODELS - VIEWS - CONTROLLERS - Heim.ifi.uio.no.” *Models-Views-Controllers*, 10 Dec. 1979, heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

[2] “Ruby on Rails.” *Wikipedia*, Wikimedia Foundation, 6 Nov. 2018, en.wikipedia.org/wiki/Ruby_on_Rails.

[3] Reenskaug, Trygve. “MVCXEROX PARC 1978-79.” *MVC XEROX PARC 1978-79*, 22 Mar. 1979, heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

[4] “World File.” *Wikipedia*, Wikimedia Foundation, 10 Nov. 2018, en.wikipedia.org/wiki/World_file.

[5] “Amazon Web Services (AWS) - Cloud Computing Services.” *Amazon*, Amazon, aws.amazon.com/.

[6] Rails. “Rails/Webpacker.” *GitHub*, 26 Oct. 2018, github.com/rails/webpacker.

[7] Foundation, Node.js. “Node JS.” *Node.js*, nodejs.org/en/.

[8] “Fast, Reliable, and Secure Dependency Management.” *Yarn*, yarnpkg.com/en/.

[9] Alexreisner. “Alexreisner/Geocoder.” *GitHub*, 12 Nov. 2018, github.com/alexreisner/geocoder.

[10] Gazay. “Gazay/Gon.” *GitHub*, 11 July 2018, github.com/gazay/gon.

[11] ohler55. “ohler55/Oj.” *GitHub*, github.com/ohler55/oj/blob/master/ext/oj/oj.c.

[12] Gimite. “Gimite/Google-Drive-Ruby.” *GitHub*, 10 Nov. 2018, github.com/gimite/google-drive-ruby.

[13] Bostock, Mike. “Data-Driven Documents.” *D3.Js*, d3js.org/.

[14] Iblue. “Iblue/d3-Rails.” *GitHub*, 30 Sept. 2018, github.com/iblue/d3-rails.

[15] Otto, Mark, and Jacob Thornton. “Bootstrap.” *Getbootstrap*, getbootstrap.com/

[16] Twbs. “Twbs/Bootstrap-Rubygem.” *GitHub*, 21 Aug. 2018, github.com/twbs/bootstrap-rubygem.

[17] Plataformatec. “Plataformatec/Devise.” *GitHub*, 23 Nov. 2018, github.com/plataformatec/devise.

[18] image-js. “Image-Js/Image-Js.” *GitHub*, 14 Nov. 2018, github.com/image-js/image-js.

[19] proj4js. “proj4js/proj4js.” *GitHub*, 24 Aug. 2018, github.com/proj4js/proj4js.

[20] “Proj4js.” *Proj4js By proj4js*, proj4js.org/.

[21] Ifttt. “IFTTT.” *IFTTT Helps Your Apps and Devices Work Together*, ifttt.com/discover.

[22] “Haversine Formula.” *Wikipedia*, Wikimedia Foundation, 24 Sept. 2018, en.wikipedia.org/wiki/Haversine_formula.

[23] “Landsat 8 Data Users Handbook - Section 1.” *Landsat Missions Timeline | Landsat Missions*, landsat.usgs.gov/landsat-8-l8-data-users-handbook-section-1.

[24] “Jordan Curve Theorem.” *Wikipedia*, Wikimedia Foundation, 24 Nov. 2018, en.wikipedia.org/wiki/Jordan_curve_theorem.

[25] “Using the USGS Landsat Level-1 Data Product.” *Landsat Missions Timeline | Landsat Missions*, landsat.usgs.gov/using-usgs-landsat-8-product.

[26] Hunsaker DJ, Pinter PJ Jr, Barnes EM, Kimball BA (2003) Estimating cotton evapotranspiration crop coefficients with a multispectral vegetation index. Irrig Sci 22:95-104

[27] “About Esri.” *Turning Analysis into Action*, www.esri.com/en-us/about/about-esri/overview.

[28] Lewis, C., & Polson, P. Theory-based design for easily-learned interfaces. Human-Computer Interaction, 1990,5,2-3, 191-220.

[29] Lewis, C., Polson, P., Wharton, C., & Rieman, J. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In Proceedings of CHI'90, (Seattle, Washington, April 1-5, 1990), ACM, New York, pp. 235-242.

[30] “Station-Specific Weather & Crop Water Use.” *SoyWater*, hprcc-agron0.unl.edu/soywater/.

[31] Rajan, N and S. Maas. 2014. Spectral crop coefficient for estimating crop water use. Ad- vances in Remote Sensing, 3(3):197-207.

[32] *International Rice Commission Newsletter Vol. 48*, FAO of the UN, www.fao.org/docrep/X0490E/x0490e06.html.