

Smart Scheduler Program

Collin Bennett and Nicholas Garde

Concept of Operations

REVISION – 2
April 30th, 2022

CONCEPT OF OPERATIONS FOR Automated Smart Scheduling

APPROVED BY:

Project Leader Date

Prof. Kalafatis Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	2/8/2022	Collin Bennett, Nicholas Garde		Draft Release
2	4/29/2022	Collin Bennett, Nicholas Garde		Final Report Revision

Table of Contents

Table of Contents	4
List of Tables	5
List of Figures	5
1. Executive Summary	6
2. Introduction	6
2.1. Background	6
2.2. Overview	7
2.3. Referenced Documents and Standards	7
3. Operating Concept	8
3.1. Scope	8
3.2. Operational Description and Constraints	8
3.3. System Description	9
3.4. Modes of Operations	10
3.5. Users	10
3.6. Support	10
4. Scenario(s)	11
4.1. College Department Staff Meeting	11
4.2. College Student Meeting	11
4.3. Business Meeting	11
5. Analysis	11
5.1. Summary of Proposed Improvements	11
5.2. Disadvantages and Limitations	11
5.3. Alternatives	12
5.4. Impact	12

List of Tables

No tables found.

List of Figures

Fig 1: System Description Block Diagram

Fig 2: Subsystem Description Diagram

1. Executive Summary

Scheduling meetings is a tedious process. It is relatively simple with two or three people, but the complexity can scale rapidly when many people are considered. Potential attendees must coordinate a time that fits into everyone's schedule, and this may not be possible. There is potential for a lot of wasted time, back-and-forth communication, and general confusion. To save these resources and maximize efficiency, our goal is to develop a semi-automated smart scheduling program that takes all the potential attendee schedules and compares them to each other. If possible, it will find the earliest available time slot(s) and schedule a meeting. There are many factors that must be accounted for, such as the importance of the person attending the meeting, the number of attendees in existing meetings, the priority of a meeting, etc. For all of these scenarios, we are designing an algorithm that finds the least disruptive time(s) to schedule a meeting and pushes these meetings to everyone's calendar.

2. Introduction

This document serves as an introduction to the Smart Scheduler, a program designed to analyze calendar data from multiple meeting attendees and schedule an optimal meeting time. If no meeting time can be found where all attendees are available, the program will find the scheduling times with the lowest cost and display any attendees that have conflicts. This scheduling solution will be designed to automate as much of the scheduling process as possible. This will save a lot of the time and manpower needed for scheduling meetings, which quickly scales in complexity as more attendees are considered.

2.1. Background

The most common method of meeting scheduling involves back-and-forth communication between attendees until a mutually available meeting time can be found. This procedure often involves one person proposing a time that works for them, and others suggesting different times, because the first does not work for them. A confusing back-and-forth of time suggestions follows until one time that works for everyone can be found. This time is then confirmed individually by all meeting attendees. Consider that this entire process may span hours and several emails, depending on how many people will attend this meeting.

Online solutions such as Doodle and Google Calendar Appointments offer a more organized form of scheduling. They eliminate the need to write emails, but hosts are still required to create meetings and each user must confirm that they are attending this meeting. Neither option provides a solution to eliminate time conflicts, and there is very little automation. Some paid solutions such as Calendar and Bookafy allow meeting organizers to view employee calendars in one place to schedule for availability, but they do not automate scheduling, a core function of our solution.

Our solution will automate the scheduling process. Meeting hosts will still be required to create a meeting and the required parameters, and attendees will need to give permission to share their calendars. These steps are very brief, and the rest of the process will be

automated. In the initial development stages, the program will give simple meeting time suggestions to the meeting organizer, but as the project progresses, machine learning will allow us to make better time suggestions based on the scheduling suggestions manually picked by a user. Our project will insert calendar events into user calendars and notify users when an event has been scheduled.

2.2. Overview

The Automated Smart Scheduler will utilize APIs and machine learning to act as an easy method for scheduling meetings with as few conflicts as possible. This will be done by creating a search algorithm to find times to schedule a meeting that minimally disrupts pre-existing meetings in attendee calendars. After this, machine learning will be used to find better scheduling solutions based on the scheduling choices that program users pick.

2.3. Referenced Documents and Standards

- “Fundamentals | Firebase Documentation,” *Google*. [Online]. Available: <https://firebase.google.com/docs/guides/>. [Accessed: 10-Feb-2022].
- IBM Cloud Education, “What is Machine Learning?,” *www.ibm.com*, Jul. 15, 2020. <https://www.ibm.com/cloud/learn/machine-learning>.

3. Operating Concept

3.1. Scope

This scheduler will make scheduling important meetings easier by taking an input of the deadline for the meeting and requesting access to all attendees' calendars to accommodate and generate a meeting with the least disruption. The finished product will achieve this by utilizing machine learning to assess the relative cost of scheduling meetings, and will push minimally disruptive schedules to each meeting's users.

3.2. Operational Description and Constraints

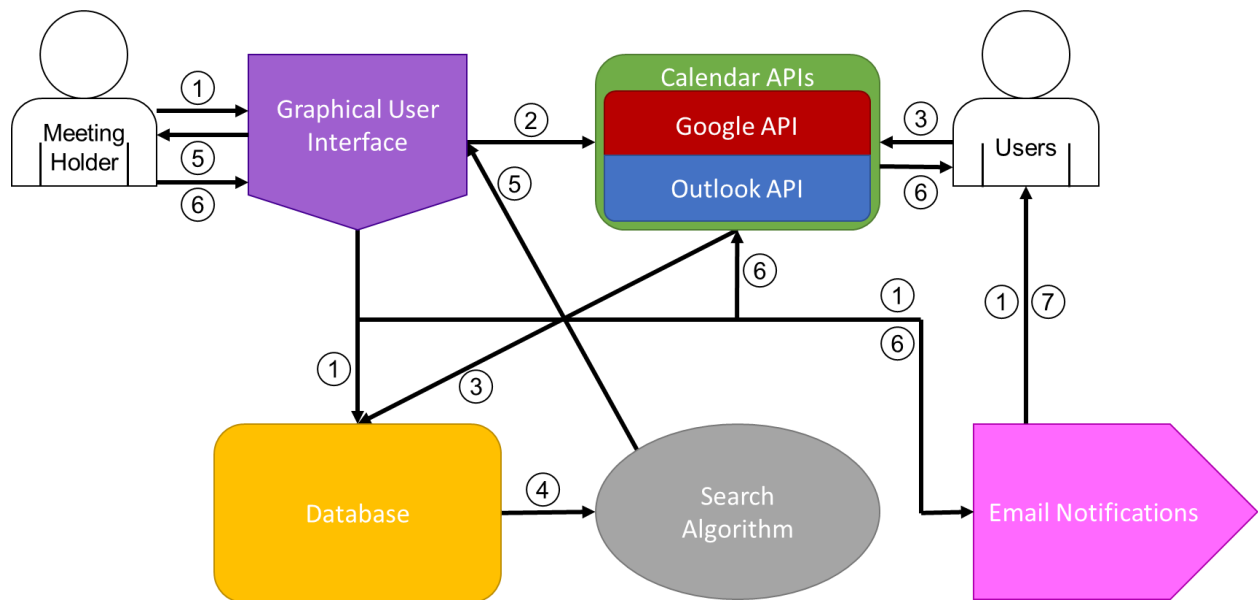


Figure 1: System Description Diagram

1. The initial user inputs the other users' contact data and meeting information, including the meeting title, description, location, and latest date for the meeting. The meeting information is stored on the database, and the meeting is shared with other users through email.
2. The meeting host shares their calendar data with the system through the Calendar APIs.
3. As users share their calendar with the system, the Calendar APIs deliver each user's calendar info to the database.
4. Once the database is sufficiently populated, the search algorithm is triggered, pulling the information stored in the database to find optimal scheduling options.
5. The search algorithm notifies the meeting host about the most optimal scheduling choices and informs them of any conflicts.
6. The meeting host decides on a meeting time, and the meeting is created on their calendar using the Calendar APIs. This meeting is then shared with the other users through their respective calendar systems, and an email notification is generated.
7. The email notification gets sent out to all participating users.

This scheduler will be used to find optimal times for scheduling meetings by finding common openings in all participant's schedules. If no openings are found, the least disruptive times will

be found and attendees with time conflicts for these times will be listed for user consideration. This scheduler is being developed with professors in mind that have many meetings and scheduling conflicts, either with their peers or their students. The current design can support meeting with an unlimited number of users, but as more users are added, the probability of unavoidable scheduling conflicts increases.

3.3. System Description

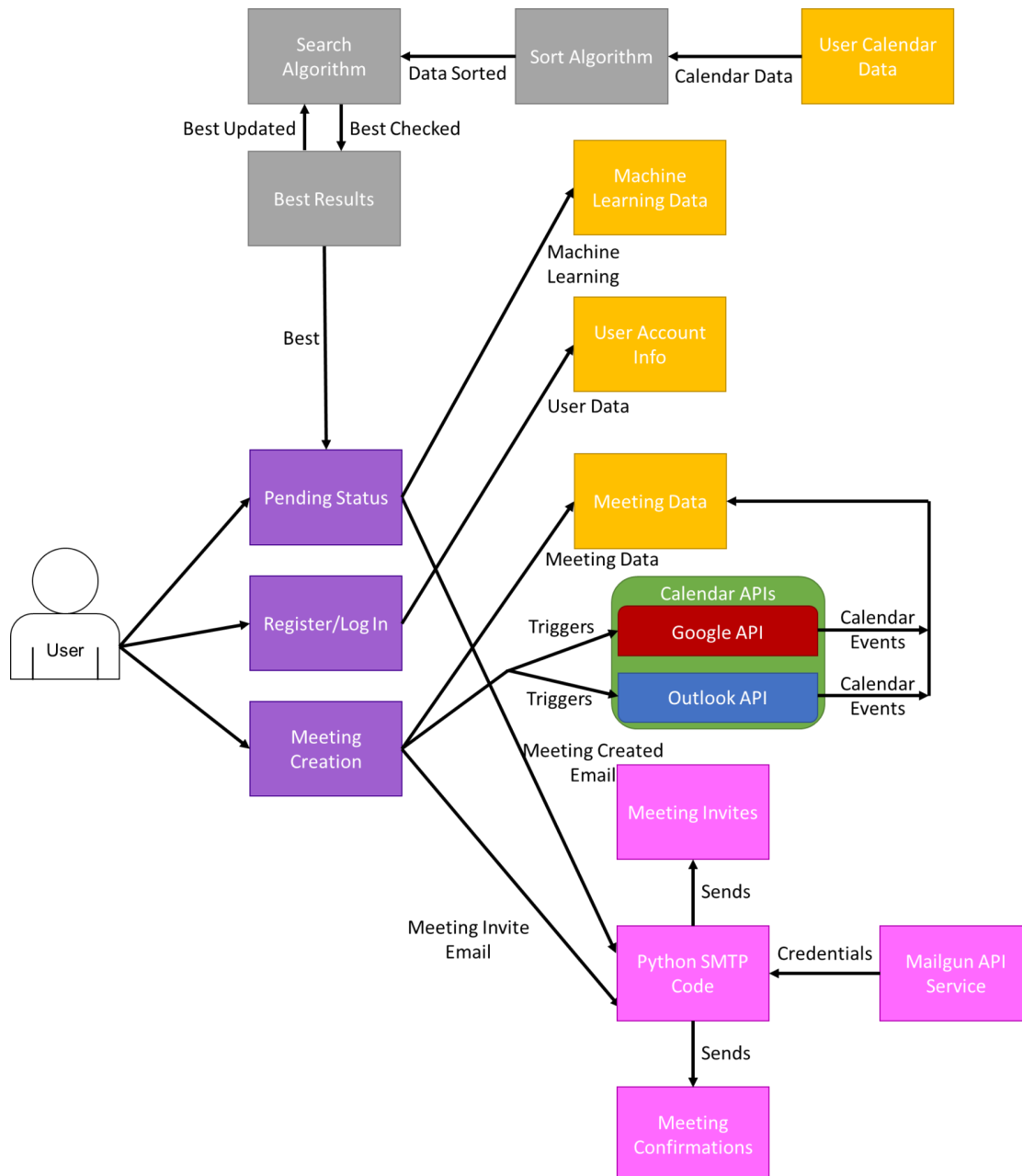


Figure 2: Subsystem Description Diagram

The Automated Smart Scheduler can be divided into five subsystems: the GUI, the search algorithm, Google Firebase, Outlook/Google APIs, and the email notification subsystem. The Graphical User Interface will allow users to input the required meeting information and schedule

a meeting later. The search algorithm utilizes the Google Calendar and Outlook APIs to extract user data from their Google and Outlook accounts respectively to store on our database. Google Firebase serves as the database solution and temporarily stores user data until it has received input from all users and is ready to be pulled and run back into the codebase for the search algorithm.

3.4. Modes of Operations

Initially, this scheduler will offer a user-specified number of minimal-cost scheduling suggestions for a program user to choose from, but as development progresses, machine learning will be used to improve these scheduling times. For example, users may prefer that meetings be held toward the middle of the day, or may want to schedule times that are not immediately after a currently scheduled meeting. The machine learning will take the user picks in account and use them to proactively determine the times a user enjoys.

3.5. Users

This scheduler is designed for professors that need to set appointments with their peers and students but often face scheduling conflicts. Manually scheduling these meetings wastes time and provides the incentive for an automated solution. The initial user base would quickly outreach from just professors to their departments, as well as their students, but this design has applications in an industry setting.

Currently, the design requires a computer with a working internet connection, email capability, and a web browser. The scheduler is being designed to be as intuitive as possible, so very little explanation is needed to fully understand how to use it. In future iterations, the design will be improved upon and the amount of data required from users will be minimized.

3.6. Support

Support for this scheduler will come from documentation as well as a README file that will be posted on its GitHub repository that will explain how to use the system. Future updates will also be uploaded to this GitHub repository containing notes on improvements. In the future, on the web app landing page, instructions for use will be provided.

4. Scenario(s)

4.1. College Department Faculty Meeting

The primary use of the Smart Scheduling System will be to schedule meetings between staff members working together within a college department. Working times range from Monday to Friday, between 8 am and 6 pm. Staff can have busy schedules, making finding a common time to meet difficult.

4.2. College Student Meeting

As development progresses, we are aiming to expand the project scope to accommodate additional groups of people, including college students. Student schedules are often sporadic and take up significant blocks throughout the day. In this regard, they are similar to college faculty schedules. There is a significant need for students to schedule study sessions, recreational events, and appointments with professors, so the software experience will be tailored to accommodate them as well.

4.3. Business Meeting

Scheduling woes are not exclusive to academia, and the same problems can be found in industry settings. Companies often have meetings within departments that require a lot of complex scheduling to ensure that everyone can attend in a timely manner.

5. Analysis

5.1. Summary of Proposed Improvements

- The system will automate the scheduling process as much as possible. Ideally, users will only need to share their calendar, email, and meeting parameters with the program.
- Elimination of no-shows/forgetfulness. The program will send all participants a notification of the scheduled meeting when it is created, serving as a reminder to attend.
- If necessary, the scheduling software will send users a breakdown of any conflicts they may have when being scheduled for a meeting.
- Meeting members will no longer need to communicate back and forth, working around each other's schedules. This will result in time saved and frustration avoided.

5.2. Disadvantages and Limitations

The Automated Smart Scheduling Program will have some limitations:

- The probability of the program being able to find a mutually available time decreases as the number of attendees increases. After a certain point, there will be no mutual times available because everyone will have filled schedules. This is not a weakness of the program, but rather an unavoidable reality of the complexity of scheduling meetings with many people.
- Due to budget and time constraints, the number of features that can be implemented in the software will be limited.

- The development of this program is very iterative. Our development team can research and plan for development milestones and timelines, but unexpected difficulties will inevitably come up. These can include software solutions not working well with each other and difficulties with machine learning.

5.3. Alternatives

Alternative solutions to the Automated Smart Scheduling System may include:

- *Meeting attendees scheduling meetings via email or based on verbal communication.* This may be simple with two or three people attempting to meet, but it quickly becomes complex when several people try to find a mutually available time, with a lot of back and forth, overhead, and time wasted.
- The web app Doodle is used to schedule meetings, but it lacks any automation features and is a paid service. It still requires users to schedule the meetings themselves. Nearly all online solutions require users to schedule the meetings themselves.
- Users send each other their calendar schedules in Google to compare for mutual meeting times. This is a completely manual version of what our program would accomplish and it is inefficient.

5.4. Impact

The Automated Smart Scheduling program has several beneficial impacts. It can save a lot of time that would otherwise be spent planning, and it keeps everyone up-to-date on schedule changes. Conversely, there are some minor privacy concerns that must be taken into consideration. This smart scheduling program has a very minimal environmental impact, but it may potentially raise privacy concerns as users must consent to sharing their calendar data with the program. This information is sensitive, so the names of events are never shared with the program, and after a meeting is scheduled, this calendar data is deleted.

Smart Scheduler Program

Collin Bennett and Nicholas Garde

Functional System Requirements

REVISION – 2
April 30, 2022

FUNCTIONAL SYSTEM REQUIREMENTS FOR Automated Smart Scheduling

APPROVED BY:

Project Leader Date

Prof. Kalafatis Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	2/23/2022	Collin Bennett, Nicholas Garde	Collin Bennett	Draft Release
2	4/30/2022	Nicholas Garde	Collin Bennett	Final Report Revision

Table of Contents

Table of Contents	16
List of Tables	17
List of Figures	17
1. Introduction	18
1.1. Purpose and Scope	18
1.2. Responsibility and Change Authority	18
2. Applicable and Reference Documents	19
2.1. Applicable Documents	19
2.2. Order of Precedence	19
3. Requirements	20
3.1. System Definition	20
3.2. Characteristics	22
3.2.1. Performance Requirements	22
3.2.2. Hardware and Software Requirements	22
3.2.3. Communication Requirements	23
3.2.4. Failure Propagation	23
4. Support Requirements	23
Appendix A Acronyms and Abbreviations	24
Appendix B Definition of Terms	24

List of Tables

Table 1: Subsystem Leads
Table 2: Applicable Documents
Table 3: Schedule
Table 4: Validation Plan

List of Figures

Fig 1: Automated Smart Scheduler Block Diagram

1. Introduction

1.1. *Purpose and Scope*

Scheduling Meetings is a tedious process. It can be relatively simple with two or three people, but the complexity can scale rapidly when many people are considered. Potential attendees must coordinate times that work for everyone. Sometimes, this is not possible. This document defines the functional system requirements for the Automated Smart Scheduler. The program is divided into five subsections: the GUI, the search algorithm, Google Firebase (database), the API communication subsystem, and the email notification subsystem. The search algorithm comprises the schedule analysis software. The API communication subsystem transfers data between Outlook, Google Calendar, and Google Firebase. Firebase holds the required user information, including account credentials and attendee calendar information. The GUI is the user's primary method of interacting with the program. This includes creating meetings and pushing out the scheduled meeting to user calendars. The email notification system ensures all meeting attendees are informed of meetings that they have been invited to and confirming when a meeting is scheduled. The verification requirements for the project are contained in a separate verification and validation plan.

1.2. *Responsibility and Change Authority*

The team leader, Collin Bennett, will be responsible for verifying all the requirements of the project are met. These requirements can only be changed with the approval of both team members and the sponsor.

Subsystem	Responsibility
Search Algorithm	Nicholas Garde
API Communication Subsystem	Nicholas Garde, Collin Bennett
Google Firebase (Database)	Nicholas Garde, Collin Bennett
Graphical User Interface	Collin Bennett
Email Notification Subsystem	Collin Bennett

Table 1: Subsystem Leads

2. Applicable and Reference Documents

2.1. *Applicable Documents*

The following documents, of the exact issue and revision shown, form a part of this specification to the extent specified herein:

Publisher	Revision/Release Date	Document Title
Python Developers	3.10.2	The Python Standard Library
Anaconda, Inc.	1.10.0	Anaconda Documentation
Streamlit, Inc.	1.5.0	Streamlit documentation
Google LLC	5.2.0	Google Firebase Documentation
Outlook Developers	16.17	Outlook Developer Documentation
Google LLC	2020.04.7	Google Calendar Documentation
Mailgun, Inc.	9.25.3	Mailgun Documentation
Manning Publications	2017	Deep Learning with Python
Scikit-learn Developers	1.0	Scikit-learn User Guide

Table 2: Applicable Documents

2.2. *Order of Precedence*

In the event of a conflict between the text of this specification and an applicable document cited herein, the text of this specification takes precedence without any exceptions.

All specifications, standards, exhibits, drawings, or other documents that are invoked as “applicable” in this specification are incorporated as cited. All documents that are referred to within an applicable report are for guidance and information only, except ICDs that have their relevant documents considered to be incorporated as cited.

3. Requirements

3.1. System Definition

The Smart Scheduler is an application for automating the scheduling of collaborative meetings. Users give permission for their calendar data to be shared with the program, and all the schedules are checked against each other for mutually available time slots. If no available times are found, the system will generate the best times to meet while notifying the meeting creator of the people that have time conflicts during the meeting time. Once a meeting is scheduled, the user will be notified. This approach is entirely software-based, but it contains several distinct subsystems, including the search algorithm, API communication subsystem, Google Firebase, Graphical User Interface, and the email notification subsystem. An overview of each subsystem can be found below.

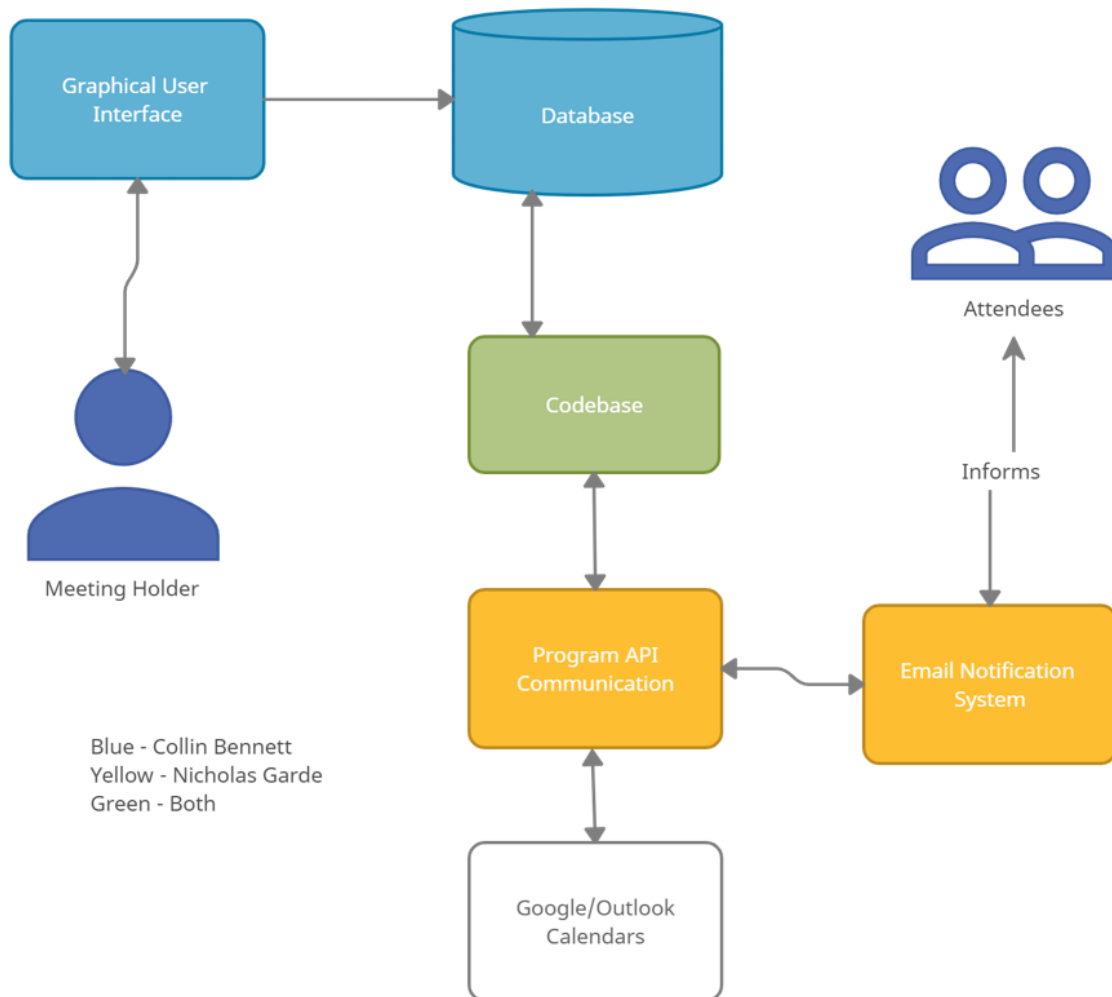


Figure 1: Automated Smart Scheduler Block Diagram

Search Algorithm: The search algorithm for the Automated Smart Scheduling Program contains the program files needed to analyze calendar data and make decisions in regards to scheduling and notifications. It forms the foundation of the scheduling solution and works with the other subsystems to deliver a functioning web application.

API Communication System: These APIs will be used for back-and-forth communication between the database and the Outlook and Google calendar services. The GUI will use the APIs to request access to a user's calendar. Once a user gives the required permissions, their calendar data is stored on the database in Google Firebase. At the end of the process, these APIs will be used to push events to both Google and Outlook attendee calendars.

Database: Google Firebase will be used as a database solution to securely store user passwords, email addresses, and schedules. New users will sign up to use the scheduler, which will save their login information to the database, and once they give calendar sharing permissions, their calendar events up to the meeting deadline will be temporarily saved to the database.

Graphical User Interface: The GUI is created using Streamlit, an app framework used for developing web applications entirely within Python. Streamlit works well with machine learning tools such as Scikit-Learn. The GUI will be the client's primary way of interacting with the scheduler. Users will be able to register, sign in, manage their account, create meetings, and send meetings to attendee calendars. After the scheduler has determined a meeting time that works, that meeting will be added to all recipient calendars, and all parties involved notified via email. In the event of a time conflict, the GUI will notify users of the attendees that have time conflicts with the proposed meeting times.

Email Notification System: Once a potential meeting is created, an email is sent to users asking them to RSVP to a meeting and share calendar information. Afterwards, when a meeting time is pushed out to attendee calendars, a confirmation email is sent.

3.2. Characteristics

3.2.1. Performance Requirements

3.2.1.1. Graphical User Interface Optimization and Simplicity

Our GUI shall be well optimized, responsive, visually pleasing, and intuitive.

Rationale: This is a very subjective performance requirement, but users will respond much more positively to a pleasant-looking GUI that is simple to use. A cluttered, confusing, and drab interface with long wait times will make for a miserable user experience. The primary goal of this scheduler is to save time, and having long wait times will render it useless.

3.2.1.2. Automated Scheduling Success

Our initial python script will take meetings from attendee Google and Outlook calendars and find mutually available meeting times between them. This is done by pulling events from each calendar for comparison. This code is undergoing strict testing with many different schedules to ensure that it gets the right information and lists available dates every single time. This is the most critical step in our program, as well as the most prone to error.

Rationale: This is the core system performance requirement. If the available meeting dates are incorrect, then scheduled meetings will result in time conflicts, rendering the entire system useless.

3.2.1.3. Efficiency and Execution Time

The Automated Scheduling system shall support a total code execution time of, at maximum, five seconds.

Rationale: The Automated Scheduling system should not run for an excessively long time, as this will result in bad user feedback and a lack of product enthusiasm.

3.2.2. Hardware and Software Requirements

3.2.2.1. Hardware Requirements

The scheduling software is not resource-intensive and can be run on any computer that can run a web browser. At a minimum, a computer running the Automated Smart Scheduler must have approximately 100MB of available hard drive space, 128MB of RAM, and an Intel Pentium 4 processor equivalent or higher.

3.2.2.2. Software Installations

To run the program locally, the computer operating system must be a version of Windows or Linux. Conda is the package manager used for Python. Streamlit, Outlook, and Google packages must be installed. These packages, documentations, and installation instructions can be found in the README file in the code repository. To run the program as a website, a web browser is needed. This is much simpler.

3.2.3. Communication Requirements

3.2.3.1. Outlook and Google Calendar APIs

Communication between the database and Outlook/Google Calendar will happen through the APIs. API calls are used to send calendar data back and forth once it has been analyzed.

3.2.3.2. Google Firebase

Google Firebase will serve as our database solution for the storage of user information including email, passwords, and calendar data in a secure, encrypted fashion. Firebase will communicate with the code base, passing data back and forth using API calls.

3.2.3.3. Email APIs

Communication with meeting attendees will happen via email using the Mailgun API and the SMTP email protocol. Users will be notified of meeting invitations and meeting confirmations.

3.2.4. Failure Propagation

The Automated Smart Scheduler will not allow propagation of failures beyond errors encountered during development. Any software failures, including the system crashing, can be rectified by restarting the system. If any errors occur, the user will be sent a report based on what part of the system failed.

4. Support Requirements

The Automated Smart Scheduler requires an active internet connection to schedule collaborative meetings. This solution is completely software-based and designed to run on computers with very low specifications. The program is not resource-intensive and can be run on any computer capable of browsing the internet. The scheduler contains everything needed to operate in the application. Meeting creators are only responsible for sharing their calendars, application login information, and required meeting parameters. Meeting attendees are only responsible for sharing their calendars and calendar emails. Should clients encounter any bugs, they can contact us directly and we will address the issue(s).

Appendix A: Acronyms and Abbreviations

GUI	Graphical User Interface
MB	Megabyte (10^6 bytes)
API	Application Programming Interface
FSR	Functional System Requirements
ICD	Interface Control Document

Appendix B: Definition of Terms

Terms will be added with later revisions. The current language is simple enough to understand.

Smart Scheduler Program

Collin Bennett and Nicholas Garde

Interface Control Document

REVISION – 2
February 23, 2022

INTERFACE CONTROL DOCUMENT FOR Automated Smart Scheduling

APPROVED BY:

Project Leader Date

Prof. Kalafatis Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	2/23/2022	Nicholas Garde	Collin Bennett	Draft Release
2	4/29/2022	Nicholas Garde	Collin Bennett	Final Revision

Table of Contents

Table of Contents	28
List of Tables	29
List of Figures	29
1. Overview	30
2. References and Definitions	31
2.1. References	31
2.2. Definitions	31
3. Codebase Resources	32
3.1. Google API Communication	32
3.2. Outlook API Communication	32
4. Database	33
4.1. Firebase Communication	33
5. Graphical User Interface	34
5.1. Streamlit Communication	34

List of Tables

Table 1: Reference Documents

List of Figures

Fig 1: Google API Authentication and Fetching

Fig 2: Outlook API Authentication and Fetching

Fig 3: Firebase API Pushing, Getting, and Deleting

Fig 4: Streamlit Tutorial Documentation

1. Overview

This document describes the references, definitions, and resources for all of the subsystems within the Automated Smart Scheduler. The Smart Scheduler can be broken down into five subsystems: Graphical User Interface subsystem, which users interact with to allow the two Calendar API subsystems to communicate with the Database Subsystem to store their calendar data, which the Search Algorithm subsystem uses to find a result and display to the users through the GUI and the Email Notification subsystems. The Graphical User Interface subsystem is the link between the users and the codebase, allowing users to send information to be read by the system. The Database subsystem is used as storage for this data in a way that it can later be accessed by the codebase. The Calendar API Subsystems are broken up into the Google Calendar and Microsoft Outlook subsystems, and they manage user connection to their data and ask for permission to access and store it through the GUI. The Search Algorithm subsystem is the heart of the codebase, compiling the data collected in the database and using it to find optimal scheduling solutions. The Email Notification subsystem is a form of output to the system by reporting to its users once a decision has been made.

2. References and Definitions

2.1. *References*

Document Title	Version	Publisher
Google Calendar API documentation	v3	Google
Outlook API documentation	v16.17	Microsoft
Firebase API for Python documentation	v5.2.0	Google
Streamlit documentation	v1.5.0	Streamlit Inc

Table 1: Reference Documents

2.2. *Definitions*

Application Programming Interface (API): method of fetching and sending user information securely through communication with Google Calendar, Outlook, and Firebase

- .json: Formatting used for sending and receiving data on Firebase
- Graphical User Interface (GUI): interface through which users can interact with the codebase

3. Codebase Resources

3.1. Google API Communication

The Google Calendar API allows for requesting user information in the form of event objects

Resource URL

<https://github.com/googleapis/google-api-python-client>

```
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build

flow = InstalledAppFlow.from_client_secrets_file(
    'client_secrets.json',
    scopes=['openid', 'https://www.googleapis.com/auth/userinfo.email', 'https://www.googleapis.com/auth/userinfo.profile'])

flow.run_local_server()
credentials = flow.credentials

service = build('calendar', 'v3', credentials=credentials)

# Optionally, view the email address of the authenticated user.
user_info_service = build('oauth2', 'v2', credentials=credentials)
user_info = user_info_service.userinfo().get().execute()
print(user_info['email'])
```

Figure 1: Google API Authentication and Fetching

3.2. Outlook API Communication

The Outlook API allows for requesting user information in the form of .csv files

Resource URL

<https://docs.microsoft.com/en-us/graph/outlook-calendar-concept-overview>

```
from O365 import Account, MSGraphProtocol

CLIENT_ID = '<your client id>'
SECRET_ID = '<your secret id>'

credentials = (CLIENT_ID, SECRET_ID)
```

Figure 2: Outlook API Authentication and Fetching

4. Database

4.1. *Firestore Communication*

The Firestore API allows for sending data to our database and calling it back when it is needed

Resource URL

<https://firebase.google.com/docs/reference/admin/python>

`get(etag=False, shallow=False)`

Returns the value, and optionally the ETag, at the current location of the database.

Parameters:	<ul style="list-style-type: none">• <code>etag</code> – A boolean indicating whether the Etag value should be returned or not (optional).• <code>shallow</code> – A boolean indicating whether to execute a shallow read (optional). Shallow reads do not retrieve the child nodes of the current database location. Cannot be set to True if <code>etag</code> is also set to True.
Returns:	If <code>etag</code> is False returns the decoded JSON value of the current database location. If <code>etag</code> is True, returns a 2-tuple consisting of the decoded JSON value and the Etag associated with the current database location.
Return type:	object
Raises:	<ul style="list-style-type: none">• <code>ValueError</code> – If both <code>etag</code> and <code>shallow</code> are set to True.• <code>FirebaseError</code> – If an error occurs while communicating with the remote database server.

`push(value="")`

Creates a new child node.

The optional value argument can be used to provide an initial value for the child node. If no value is provided, child node will have empty string as the default value.

Parameters:	<code>value</code> – JSON-serializable initial value for the child node (optional).
Returns:	A Reference representing the newly created child node.
Return type:	<code>Reference</code>
Raises:	<ul style="list-style-type: none">• <code>ValueError</code> – If the value is None.• <code>TypeError</code> – If the value is not JSON-serializable.• <code>FirebaseError</code> – If an error occurs while communicating with the remote database server.

`delete()`

Deletes this node from the database.

Raises:	<code>FirebaseError</code> – If an error occurs while communicating with the remote database server.
---------	--

Figure 3: Firestore API Pushing, Getting, and Deleting

5. Graphical User Interface

5.1. *Streamlit*

The Streamlit Framework allows for quick visualization of Python code and straightforward user-interface creation

Resource URL

<https://docs.streamlit.io/>

```
import streamlit as st

left_column, right_column = st.columns(2)
# You can use a column just like st.sidebar:
left_column.button('Press me!')

# Or even better, call Streamlit functions inside a "with" block:
with right_column:
    chosen = st.radio(
        'Sorting hat',
        ("Gryffindor", "Ravenclaw", "Hufflepuff", "Slytherin"))
    st.write(f"You are in {chosen} house!")
```

Figure 4: Streamlit Tutorial Documentation

Smart Scheduler Program

Collin Bennett and Nicholas Garde

Execution Plan

REVISION – 2
April 30, 2022

Validation Plan for Smart Scheduler Program

Complete	
In Progress	
Incomplete	

Task	2/9/22	3/3/22	3/10/22	3/17/22	3/24/22	3/31/22	4/7/22	4/14/22	4/21/22	4/28/22
Initial Research										
Explore Database Options										
Explore Email API Solutions										
Create Prototype GUI										
Begin Search Algorithm										
Learn Google API Calls										
Learn Outlook API Calls										
Learn Google Firebase API Calls										
GUI Development Progress										
Choose Email API Solution										
Have Google API Functionality Working										
Have Outlook API Research Done										
Begin Implementing GCal into Algorithm										
Integrate Database With GUI										
Create and Plan Email Scenarios										
Incorporate Google Calendar into Scheduler										
Continue with Google Calendar integration										
Integrate Database with Search Algorithm										
GUI event scheduling in Firebase										
Program Emails										

Integrate Search Algorithm with GUI										
Final Presentation										
Demo										
Final Report										
Beta Testing Over Summer										

Smart Scheduler Program

Collin Bennett and Nicholas Garde

Validation Plan

REVISION – 2
April 30, 2022

Validation Plan for Smart Scheduler Program

Complete	
In Progress	
Incomplete	

Task	Specification	Method	Result	Owner
Google Calendar – Pull Events	Testing event fetching accuracy of Google Calendar	Use Google's OAuth2.0 to request data from the user from their calendar	Complete	Nicholas Garde
Google Calendar - Insert Events	Google API is properly inserting events	Ensure that calendar events can be added to specified use calendars without error	Complete	Nicholas Garde
Outlook Calendar – Pull Events	Testing event fetching accuracy of Outlook Calendar	Use Outlook's Authentication to request data from the user from their calendar	Complete	Collin Bennett
Outlook Calendar - Insert Events	Outlook API is properly inserting events	Ensure that calendar events can be added to specified use calendars without error	Complete	Collin Bennett
Firebase – User Authentication	Firebase is storing user account details	When people sign up as an admin or user for the scheduler, ensure their account information is properly stored	Complete	Collin Bennett
Firebase – User Credentials	Firebase is storing user credentials	Ensure user credentials collected on signup are stored in Firebase	Complete	Collin Bennett
Firebase – Email List	Firebase is storing user email lists	Ensure users can add contacts correctly to their email lists, and that these are stored in Firebase	Complete	Collin Bennett
Firebase – Meetings	Firebase is storing meetings	Ensure Firebase is properly holding user meetings and associated parameters	Complete	Collin Bennett
Firebase – User Calendars	Firebase is storing user calendars	Verify Firebase is holding required attendee events after OAuth	Complete	Nicholas Garde
Firebase – ML Data	Firebase is storing Machine Learning Data	Verify Firebase is holding data to be used for machine learning next semester after a scheduling time is chosen	Complete	Nicholas Garde

Search Algorithm – Sort Data	Ensure user events are properly sorted	Pulling the data from firebase to be sorted onto a data structure to be searched through next	Complete	Nicholas Garde
Search Algorithm – Search	Use rules to define optimality of solutions	Searching through aforementioned data structure to find optimal time to schedule a meeting	Complete	Nicholas Garde
Email API – Invitations	Ensure invitations are sent	Verify attendees are receiving invite emails with the correct meeting information	Complete	Collin Bennett
Email API – Confirmations	Ensure confirmations are sent	Verify attendees are receiving confirmation emails with the correct meeting information	Complete	Collin Bennett
GUI – Ease of Use	Exploring GUI Navigation ease-of-use	Ensure the GUI navigation works properly, nothing is out of place, and error messages are displayed	Complete	Collin Bennett
GUI – Change Name	User name change	User function to change name works properly	Complete	Collin Bennett
GUI – Change Password	User password reset	User function to change the password sends an email and works properly	Complete	Collin Bennett
GUI – Delete Account	User delete account	User function to delete account works properly	Complete	Collin Bennett
GUI – Create a Meeting	Meeting creation success	GUI screen to create a meeting works properly and has no bugs	Complete	Collin Bennett
GUI – View Pending Invites	Pending invitation success	GUI screen to view pending invites works properly and has no bugs	Complete	Collin Bennett
Bug Fixing	Find and fix bugs	Spend the time before demo running through our program, trying to find any hidden bugs	Complete	Both
Beta Testing	Summer sponsor feedback	Hand the program over to our sponsor who will supply us with feedback and methods to improve	In Progress	Dr. Reddy

Performance on Execution Plan

The execution plan was completed in its entirety. Originally, we were planning to have Outlook functionality implemented in addition to Google by the end of ECEN 403, but our sponsor decided that we should do extensive testing and have a fully working product before integrating another calendar service into our design. This allowed us to reduce our scope slightly for the spring semester. The research and testing we have completed for Outlook Calendar will be implemented in the fall semester.

Performance on Validation Plan

The validation plan was also completed in its entirety. All subsystems are behaving as expected, and have been subjected to rigorous design checks. The validation plan was completed thoroughly, with each aspect of the individual subsystems being rigorously checked.

Smart Scheduler Program

Collin Bennett and Nicholas Garde

Subsystem Reports

REVISION – 1
April 30, 2022

SUBSYSTEMS REPORT FOR Automated Smart Scheduling

APPROVED BY:

Project Leader
Date

Prof. Kalafatis	Date
-----------------	------

T/A
Date

Change Record

Rev.	Date	Originator	Approvals	Description
1	4/30/2022	Collin Bennett, Nicholas Garde		Final Report Revision

Table of Contents

Table of Contents	45
List of Tables	47
List of Figures	47
1. Introduction	48
2. Graphical User Interface Subsystem Report	49
2.1. Subsystem Introduction	49
2.2. Subsystem Details	49
2.3. Subsystem Validation	57
2.4. Subsystem Conclusion	58
3. Search Algorithm Subsystem Report	58
3.1. Subsystem Introduction	58
3.2. Sort Algorithm	58
3.2.1. Operation	58
3.2.2. Validation	60
3.3. Search Algorithm	60
3.3.1. Operation	60
3.3.2. Validation	61
3.4. Subsystem Conclusion	62
4. Google Calendar and Outlook API Subsystem Report	62
4.1. Subsystem Introduction	62
4.2. Google Calendar Pull	62
4.2.1. Operation	62
4.2.2. Validation	64
4.3. Google Calendar Push	65
4.3.1. Operation	65
4.3.2. Validation	65
4.4. Outlook Calendar - Pull Events	65
4.4.1. Operation	65
4.4.2. Validation	67
4.5. Outlook Calendar Push	67
4.5.1. Operation	67
4.5.2. Validation	67
4.6. Subsystem Conclusion	68

5. Database Subsystem Report	68
5.1. Subsystem Introduction	68
5.2. Firebase Push	69
5.2.1. Operation	69
5.2.2. Validation	69
5.3. Firebase Pull	70
5.3.1. Operation	70
5.3.2. Validation	70
5.4. ML Push	70
5.4.1. Operation	70
5.4.2. Validation	70
5.5. Account Data	71
5.5.1. Operation	71
5.5.2. Validation	73
5.6. Subsystem Conclusion	73
6. Email Subsystem Report	74
6.1. Subsystem Introduction	74
6.2. Mailgun and SMTP	75
6.2.1. Operation	75
6.3. Meeting Invitation Email	76
6.3.1. Operation	76
6.3.2. Validation	77
6.4. Meeting Confirmation Email	77
6.4.1. Operation	77
6.4.2. Validation	78
6.5. Subsystem Conclusion	78

List of Tables

List of Figures

Fig 1: GUI Subsystem Block Diagram
Fig 2: Registration Page
Fig 3: Login Page
Fig 4: Change Name Page
Fig 5: Change Password Page
Fig 6: Delete Account Page
Fig 7: Adding Contacts
Fig 8: Appointment Details Page
Fig 9: Join a Meeting Page
Fig 10: Pending Member Status
Fig 11: Meeting Times
Fig 12: Search Algorithm Subsystem Block Diagram
Fig 13: Code snippet - user information
Fig 14: Code snippet - search algorithm
Fig 15: Code snippet - criteria
Fig 16: Code snippet - best line resort
Fig 17: API Subsystem Block Diagram
Fig 18: OAuth 2.0 Request Page
Fig 19: OAuth2.0 Permissions
Fig 20: API Call Code Snippet
Fig 21: Data selection and storage code snippet
Fig 22: GCal Push code snippet
Fig 23: Outlook permissions request menu
Fig 24: Outlook pulling all calendar events
Fig 25: Outlook pushing a calendar event
Fig 26: Database Subsystem Block Diagram
Fig 27: Firebase Push storing user events onto the database
Fig 28: ML Push data stored in the database
Fig 29: User credentials stored in the database
Fig 30: User email list stored in the database
Fig 31: User meeting information stored in the database
Fig 32: Email Subsystem Block Diagram
Fig 33: Email code, body message
Fig 34: Email code, SMTP relay
Fig 35: Meeting invitation email example
Fig 36: Meeting confirmation email example

1. Introduction

The Smart Scheduler program will temporarily collect user calendar data and generate a meeting time that can be attended by all potential members. To begin, a meeting creator generates a potential meeting with details and a deadline and sends these details to an invited list of potential attendees through the use of the email API. Users will respond to the email and be redirected to the graphical user interface, where they can view any meetings that they have been invited to and decide if they want to RSVP or decline.

If a potential attendee accepts, they are redirected to a Google Authentication screen that requests read and write access to their calendars. Once these permissions are given, the user's calendar events up to the meeting deadline are stored in Firebase. The meeting organizer can view the status of potential attendees. This will be displayed as pending, accepted, or declined. Once the organizer decides to schedule the meeting based on the invite statuses, all calendar data stored in firebase is used to generate potential meeting times. The organizer can pick from these times, and push a meeting to attendee calendars. This program is divided into five subsystems, including the search algorithm, graphical user interface, Outlook and Google Calendar APIs, email notification subsystem, and Firebase database.

2. Graphical User Interface Subsystem Report

2.1. Subsystem Introduction

The Graphical User Interface is designed to allow users to create and schedule meetings. The subsystem has been programmed in Python using Streamlit as a framework. Streamlit allows programmers to quickly build web apps using python scripts and does not require much knowledge of frontend web development. The power subsystem was tested to confirm its stability, capacity, and consistency. Each test helped verify that the power subsystem will support the components of the system that will be powered by it.

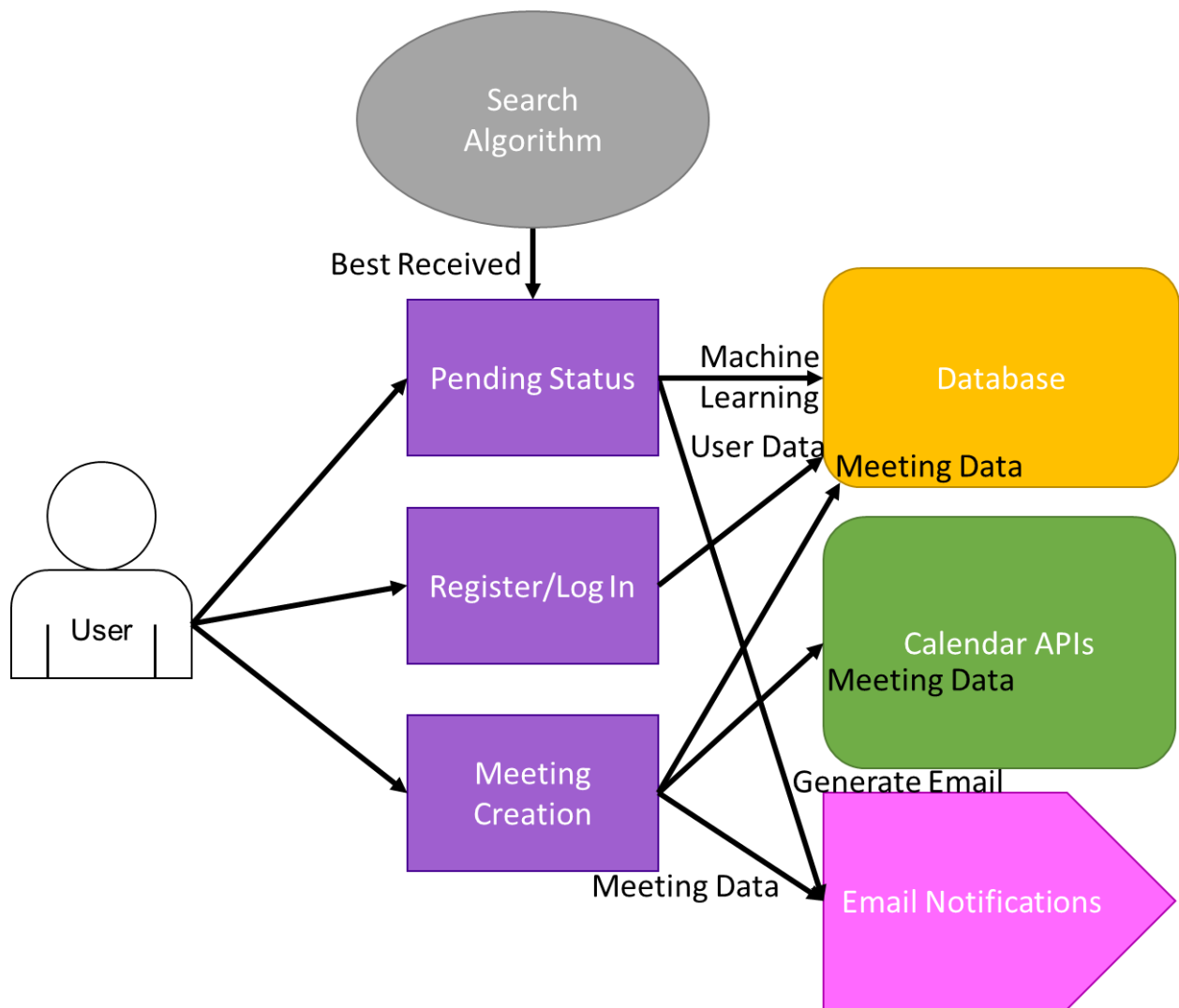
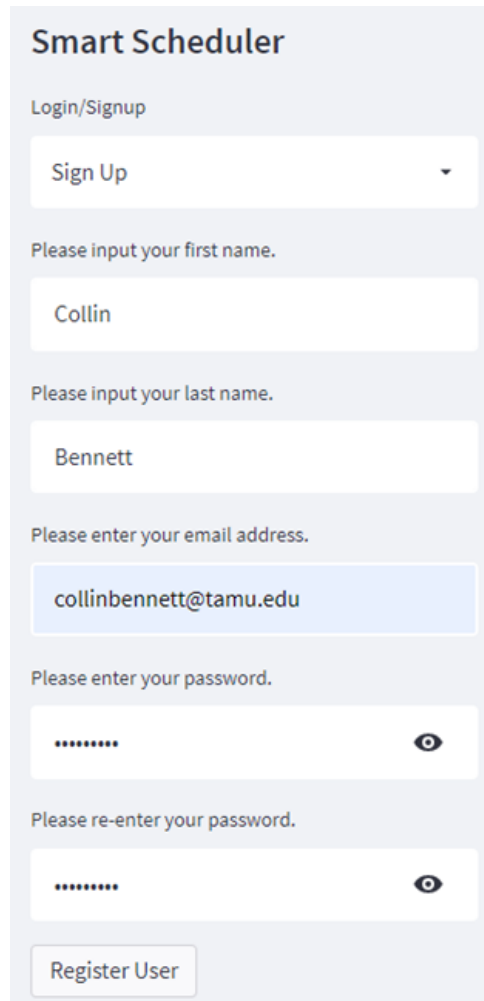


Figure 1: GUI Subsystem Block Diagram

2.2. Subsystem Details

A block diagram of the Graphical User Interface can be seen below.



The image shows a web form titled "Smart Scheduler" for user registration. It includes a "Login/Signup" dropdown menu with "Sign Up" selected. Below are input fields for "first name" (Collin), "last name" (Bennett), "email address" (collinbennett@tamu.edu), and "password" (masked with dots). A "Register User" button is at the bottom.

Smart Scheduler

Login/Signup

Sign Up ▼

Please input your first name.

Collin

Please input your last name.

Bennett

Please enter your email address.

collinbennett@tamu.edu

Please enter your password.

.....

Please re-enter your password.

.....

Register User

Figure 2: Registration Page

Users will interact with the smart scheduling program exclusively through the Graphical User Interface. Registration is the first step a user must complete. Sign up requirements include a first name, last name, valid email address, and a password at least six characters long. When the button “Register User” is pressed, these credentials are used to create an account using Google Authentication, and the user data is uploaded to Google Firebase. If any of these parameters are entered incorrectly or left blank, conditional statements in the code will catch the error and prompt the user to correct any mistakes before proceeding.

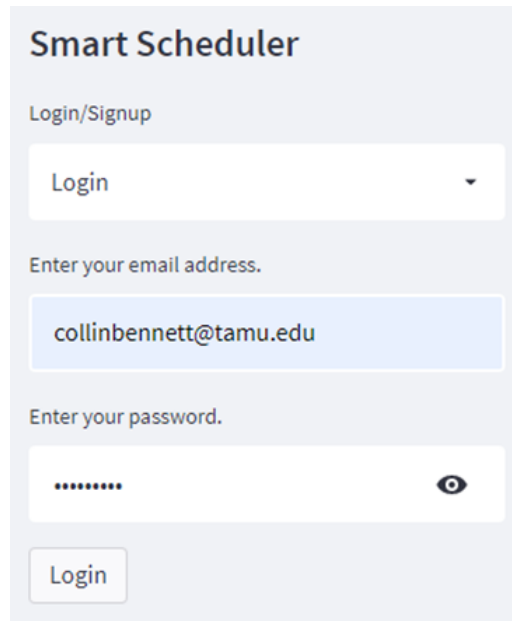
The image shows a web interface for 'Smart Scheduler'. At the top, the title 'Smart Scheduler' is displayed in a bold, dark blue font. Below the title, the text 'Login/Signup' is shown in a smaller, grey font. There is a dropdown menu with 'Login' selected. Below this, the prompt 'Enter your email address.' is followed by a text input field containing the email 'collinbennett@tamu.edu'. The next prompt is 'Enter your password.', followed by a password input field with masked characters (dots) and an eye icon to toggle visibility. At the bottom, there is a 'Login' button.

Figure 3: Login Page

After successful account creation, users will be prompted to login. This interface works very similarly to the registration page, but only requires an email and password. Like the registration page, if any field is left blank or incorrectly entered, the program will alert the user that the email address or password is incorrect. If the email and password are successfully entered, Google Authentication will log the user into the application, allowing them to use any of the following features.

Change Your Name

Enter your new, updated name below.

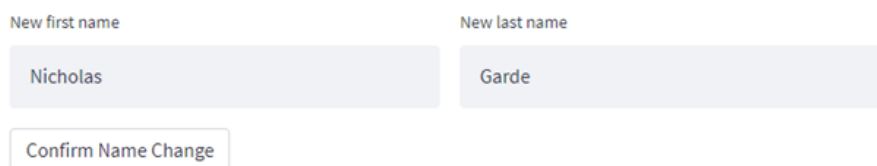
The image shows a form titled 'Change Your Name'. It has two input fields: 'New first name' with the value 'Nicholas' and 'New last name' with the value 'Garde'. Below these fields is a button labeled 'Confirm Name Change'.

Figure 4: Change Name Page

If an account holder would like to change his or her name, this function in the dropdown menu will allow them to do so. Users will enter a new first name and a new last name, and once the “Confirm Name Change” button is pressed, the new name will be uploaded to Google Firebase. When potential meeting attendees are sent email notifications, this name will be used in reference to the meeting.

Change Password

Would you like to reset your password?

Confirm password reset

Figure 5: Change Password Page

This simple account feature will allow users to change the password used to login to his or her account. If “Confirm password reset” is pressed, the user will be sent an email by Google Authentication prompting them to change their password.

Delete Account

Are you sure you would like to delete your account? It is unrecoverable.

Confirm Account Deletion

Figure 6: Delete Account Page.

This function allows users to delete their Smart Scheduler account. This button, “Confirm Account Deletion” removes the user from Google Authentication records, and removes any of their saved data, including pending meetings, email addresses, and any remaining personal information.

Optional - Add contacts to contact list.

Enter a full name.	Enter an email.
<input type="text" value="John Smith"/>	<input type="text" value="johnsmith@tamu.edu"/>
<input type="button" value="Enter into contacts."/>	
John Smith added to contact list.	

Figure 7: Adding Contacts.

When a Smart Scheduler account holder wishes to invite a person to a meeting, they must have their name and contact email address so a notification email can be sent out. The Adding Contacts feature allows users to add contacts to a personal address book for easy access later during meeting creation. This feature works by uploading the name and email address pair to Google Firebase under an email list that is only available to the user that is currently signed in.

Enter appointment details.

Enter the name of the meeting.

Study Session

Enter a meeting description.

Study for the CSCE 350 Computer Architecture Exam

Enter a meeting deadline.

2022/05/20

Meeting Duration - Hours

3

-

+

Meeting Duration - Minutes

30

-

+

Choose meeting attendees.

John Smith X

User One X

User Two X

User Three X

User Four X

X ▼

johnsmith@tamu.edu appended to the email list.

smartschedulercuser1@gmail.com appended to the email list.

smartschedulercuser2@gmail.com appended to the email list.

smartschedulercuser3@gmail.com appended to the email list.

smartschedulercuser4@gmail.com appended to the email list.

Enter a virtual meeting link or physical location. (Optional)

Texas A&M Campus, Zachry Building, Room 134

Generate Meeting and Invite Attendees

Figure 8: Appointment Details Page.

Once a user is logged in and has added the required invitees to their address book, they may begin the process of scheduling meetings by entering the necessary appointment details. These details include:

- **Meeting name**, a required parameter that will be used to identify the meeting
- **Meeting description**, an optional parameter that can be used to provide context to the purpose of the meeting
- **Meeting deadline**, a required parameter with which a user will specify the absolute last day a meeting can be held on. Meetings can be scheduled between the hours of 8am and 6pm
- **Meeting duration – hours**, a required parameter that determines the hour length of the created meeting
- **Meeting duration – minutes**, a required parameter that determines the minute length of the created meeting. For example, a created meeting may have a time duration of 1 hour and 45 minutes
- **Choose meeting attendees**, a required parameter that allows the user to select multiple potential attendees by name. Once these names are selected, the contact emails associated with these them are appended to an email list that is held by the program until the submit button is pressed.
- **Meeting Link/Physical Location**, an optional parameter that allows users to insert a location or zoom link.

Once all of these details are inserted and the “Generate Meeting and Invite Attendees” button is pressed, the program will ensure that no required fields were left blank and notify the user if any are. If everything is properly inserted, calendar data will begin to be pulled, starting with the meeting organizer’s data. This is documented in further detail below, but once the button is pressed, the organizer is redirected to a Google Authentication screen asking for permission to access their calendar data. Once they consent, the program will continue, and emails will be sent to all invitees in the email list.

Join a Meeting

Smart Scheduler does not store meeting names, only meeting times. Once a meeting is scheduled, all of your information is deleted from our databases.

Please enter your contact email address. This is the address an email was just sent to.

johnsmith@tamu.edu

Please enter your Google Calendar Email address.

johnsmith@tamu.edu

View my meeting invitations

Meeting: Study Session

Meeting deadline: 05/20/2022

Meeting host: Collin Bennett

Share calendar and RSVP

Decline meeting

Figure 9: Join a Meeting Page.

Once an organizer has created a meeting, users will receive a notification email. This process is explained in detail in the Email subsystem report. This email will contain a link to the portion of the Smart Scheduler application that allows users to join meetings, shown in Figure 9 above. Users will input their contact email (the email address an email was just sent to) and their Google Calendar email address, which is used to push a meeting to the attendee calendar. Once the “View my meeting invitations” button is pressed, the program will parse through every pending meeting on Google Firebase and find those that the specified user has been invited to.

These meetings are listed in alerts containing the meeting name, meeting deadline, and meeting host. The user can decide if they would like to share calendar information and RSVP, or decline attending the meeting. Accepting a meeting redirects the user to a Google Authentication page asking the user to share calendar data. This is the same screen seen by the meeting creator when the meeting creation button is pressed. Once the user’s calendar data is shared, there is no further action required on their part, and the status of the user changes from “Pending” to “Attending”. If a user declines the meeting invitation, their status is changed from “Pending” to “Not Attending”. No calendar data is requested from the user.

Pending Member Status

Study Session

johnsmith@tamu.edu	Attending
smartscheduleruser1@gmail.com	Attending
smartscheduleruser2@gmail.com	Attending
smartscheduleruser4@gmail.com	Pending
smartscheduleruser3@gmail.com	Declined

Enter the number of rescheduling suggestions. ?

1 3 10

SCHEDULE MEETING

Figure 10: Pending Member Status.

At any time after creating a meeting, a Smart Scheduler account holder can check the status of their pending invitations. On the Pending Member Status page, seen in Figure 10, each pending meeting is listed, with the potential attendees of each listed underneath. Each attendee will have one of three status indicators; Attending (green), Pending (yellow), or Declined (red). The attending status is determined by the existence of the attendee's shared calendar data under the meeting in Google Firebase. Once a user is ready to schedule the meeting, they can pick the number of scheduling suggestions they would like to receive using the slider below the meeting. For now, this number will range between one and ten.

Pick a preferred meeting time.

Thursday May 05, 2022 @ 8:00am

Pick meeting time

Thursday May 05, 2022 @ 8:15am

Pick meeting time

Thursday May 05, 2022 @ 8:30am

Pick meeting time

Thursday May 05, 2022 @ 8:45am

Pick meeting time

Thursday May 05, 2022 @ 9:00am

Pick meeting time

Figure 11: Meeting Times.

Once “SCHEDULE MEETING” is pressed, the search algorithm subsystem (discussed in more detail below) will find the specified number of mutually-available meeting times that can be selected by the user. If any time conflicts are found by the program during the listed times, the attendee with a conflict will be listed below the respective time with an indication of the conflict. Once a meeting time is decided upon and the “Pick meeting time” button is selected, the organizer is redirected to a second and final Google Authentication screen allowing them to push the calendar event to all attending members.

Once this is done, all attending members are sent meeting confirmation emails. As a final privacy measure, the meeting and all stored user calendar events are deleted from Firebase. In the future, the scheduling suggestions will need to be improved upon, so we are collecting the chosen times and saving them in Firebase, but these do not contain any user information.

2.3. Subsystem Validation

After extensive testing and validation, the graphical user interface was found to be working correctly. The first validation test was very subjective, and involved improving the intuitiveness and ease-of-use of the subsystem. The GUI would be used by our sponsor and TA, and both would give feedback on design choices that he liked, and design choices that he either did not like, or felt the need to remove.

The second validation test involved successfully changing the account holder’s name. This is done on the “Name Change Page”. The login name was changed multiple times, and each time, the name was found to change successfully in Google Firebase, allowing for emails to be sent with the new meeting host name. The “Change Password” and “Delete Account” pages also worked correctly. The change password function properly sends an email to the user email stored in Firebase and allows users to change their password securely. This was validated by changing the password and attempting to login with the new credentials. The delete account button successfully removes the user account from the Google Authentication service and removes the specified user information from the realtime database hosted on Firebase.

Creating a meeting also underwent rigorous testing. Because there are several meeting parameters that users can enter data into, checks were put in place to ensure that no required fields were left blank or filled with invalid data. In the event that these fields are improperly interacted with, the GUI will prompt the user to correct the specified mistake before the program is allowed to proceed.

The screen that displays invitation status also underwent extensive validation. The system displays meetings sequentially with attendee statuses under each meeting. There is a button for scheduling a meeting that once pressed, will display the desired number of meetings according to the input slider, which was proven to work correctly, every time.

2.4. Subsystem Conclusion

Each part of the subsystem was tested and found to work correctly. The GUI allows users to register, sign in, create meetings, schedule meetings, delete accounts, and change passwords. The GUI combines with the other subsystems to send emails, read and write from the database, and push and pull events from Google Calendar.

3. Search Algorithm Subsystem Report

3.1. Subsystem Introduction

The Search Algorithm subsystem is meant to first organize user data in a way that it can be parsed through before actually parsing through the data using methods that maximize its efficiency. This subsystem takes inputs from the GUI in the form of a meeting deadline class that holds the last possible date that a meeting could be scheduled, the name of the meeting, and the planned meeting duration, as well as the input of how many outputs the user would like to see. The output of this subsystem goes back to the GUI in the form of the best possible meeting times based on the provided information as well as the retrieved user calendar data stored in the database.

3.2. Sort Algorithm

3.2.1. Operation

The sorting part of this subsystem begins by constructing a calendar data structure that contains the weekdays between the day it was activated and the last possible meeting day. Inside each of these days, the calendar holds 15-minute increments for the times 8 am - 6 pm shown below. After a calendar is constructed, it needs to be populated, so the subsystem requests a Firebase Pull from the database subsystem which provides it with all of the user data that pertains to the meeting it is attempting to schedule. This data is all placed onto the calendar based on their time and date data leaving it properly sorted, and finally, the calendar is outputted to the search algorithm.

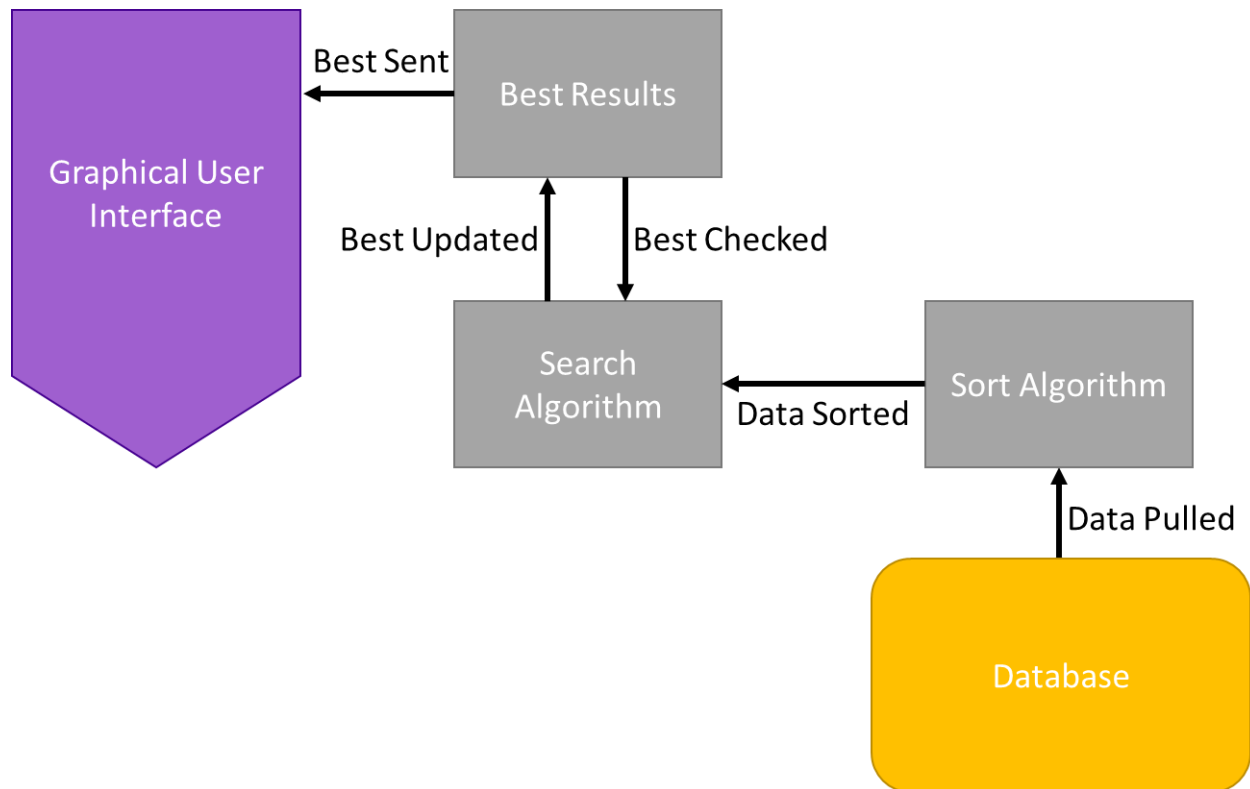


Figure 12: Search Algorithm Subsystem Block Diagram

```

# sched holds each person's schedule in the array
for sched in arryschedules:
    # event holds each event out of a given person's schedule
    for event in sched.apieventlist:
        # store the date, start time, and end time into variables
        # event is in format: 2022-03-22T15:55:00-05:00
        # pos 0-4 stores the year, 5-7 the month, 8-10 the day
        ev_date = datetime.date(int(event.start[0:4]), int(event.start[5:7]), int(event.start[8:10]))
        # pos 11-13 stores the hour, 14-16 the minute
        ev_stime = int(event.start[11:13]+event.start[14:16])
        ev_etime = int(event.end[11:13]+event.end[14:16])
        # day stores [date, [[time, [names], [summary]]]]
        for day in cal:
            # check if the event date is the current the same date in the calendar
            if ev_date == day[0]:
                # once in the correct date, check through that date's times to look for all conflicting times
                for time in day[1]:
                    # after we are completely past the event, break to stop iterating through times
                    if ev_etime < time[0]:
                        break
                    # when ev start is < l[0]+15 and ev end is > l[0] add name and summary
                    elif (ev_stime < time[0]+15 and ev_etime > time[0]):
                        # add name and that name's cost
                        time[1].append([sched.user, sched.cost])
                        # add summary
                        time[2].append(event.summary)
                        # increase the cost of rescheduling an event from this user
                        sched.cost+=1
                # Once we found and finished looking through the correct day, break to move to the next event
                break
  
```

Figure 13: Code snippet showing the placing of user information onto the calendar at times specified by their event data

3.2.2. Validation

The first step of validation for the sort algorithm is ensuring that a calendar is being properly created with empty lists under the times 8 am - 6 pm and containing all of the relevant days between the execution date and the last date and outputting it to the calendar. Once the sort algorithm correctly creates the calendar, its next test of validation is correctly executing a Firebase Pull connection with the Database subsystem. This was proved by printing the array of data retrieved to the terminal. Finally, after this was done successfully, the last step of validation was proving that the sort algorithm was adding the data onto the correct parts of the calendar by checking the start and end times of every event. All of these tests were passed by the sort algorithm, and the results can be passed directly into the search algorithm.

3.3. Search Algorithm

3.3.1. Operation

The search algorithm's goal is to pan through the data provided by the sort algorithm and find the most viable times to schedule a meeting based on its own scoring criteria. The scoring criteria of this algorithm values the number of conflicts at a given time the most and values the number of attendees being displaced by rescheduling a meeting next. Using these criteria, the search algorithm blocks out a chunk of time measured by the length of the meeting that it is trying to schedule and scores the chunk. Every time a chunk is scored it gets compared to a list of best options, where if the chunk is better than any of them, it replaces the worst scoring time on the list and the whole list gets sorted again. Once a whole day of times are scored and the best of them are placed onto the best list, the process repeats for the next day, keeping the same best list in an effort to only output the absolute best times found in the calendar. Once the search algorithm has parsed through the entire calendar, it outputs the list of best times found to the GUI for the user to select their choice.

```
# increment from left to right adding up scores
while left<=right:
    # score[0] adds the value of the length of a timetable at a given time's name array to result in how many people have a conflict at that time
    score[0]+=len(day[1][left][1])
    # This adds all of the number of attendees of every meeting that is occurring at a given time to add to score[1]
    for participants in day[1][left][2]:
        # score[1] holds the total number of attendees that would be displaced if we were to ask to reschedule any meetings at a given time
        score[1]+=int(participants)
    # this adds every user's name that has a conflict at a given time
    if(len(day[1][left][1])>0):
        for name in day[1][left][1]:
            if(not(name[0] in names)):
                names.append(name[0])
            # name[1] currently holds the value for the user name[0]'s free/busy score
            # later scoring will use this as well
    left+=1
```

Figure 14: Code snippet displaying how the search algorithm pans through a chunk of data and scores it by the number of conflicts and number of potentially displaced participants at a given increment

```

if score[0]<best[index][1][0] and (today<datetime.datetime(
    # currently best is set to length 3 and is constantly s
    # replace best[2] date with the better scoring date
    best[index][0]=day[0]
    # replace best[2] score with the better scoring score
    best[index][1]=score
    # replace best[2] time with the better scoring time
    best[index][2]=day[1][left_set][0]
    # replace best[2] names with the better scoring names
    # only add each name once
    best[index][3]= no_repeat(names, []) # was names # was

```

Figure 15: Code snippet displaying criteria for popping a value from the best list and replacing it with the chunk that scored better. The second part of the if statement is an extra check that only allows values that are after the execution of the code to be added to the best list.

```

# resort best, first by attendees displaced, then by number of user conflicts
best.sort(key=lambda x: x[1][1])
best.sort(key=lambda x: x[1][0])

```

Figure 16: Code snippet displaying how the best list gets resorted every time a chunk is placed onto it so the final value in the array is always the worst scoring one (and will be the next one removed). Sorting twice means that the first sort is almost completely overridden by the second, except when the second results in two or more identical scores, where it will keep them in order of the first. This achieves a 2-criteria sort.

3.3.2. Validation

The search algorithm only has one validation test done with two different data sets. When given a full calendar with no 0 cost times to schedule a meeting, it needs to output the best option of times where the least number of users have another meeting and where a concurrent meeting has the least number of people involved to allow it to more easily be rescheduled. When given a calendar with few holes where a meeting can fit, the algorithm needs to display these at the highest priority before suggesting any options where another user has another meeting. Both of these validation tests are passed by the search algorithm and the results are properly being output to the GUI.

3.4. Subsystem Conclusion

This subsystem has a lot to do, and truly is the core of what the Smart Scheduler is attempting to achieve. This subsystem is in charge of actually finding the best meeting times given the user data, and it currently is performing both the sort and search perfectly. In future iterations of this project, the hope is that user input, as well as the data collected by ML Push in the database subsystem, will help tune scoring criteria to provide users with even better suggestions in the future.

4. Google Calendar and Outlook API Subsystem Report

4.1. Subsystem Introduction

The Google/Outlook Calendar subsystem is an in-between that takes user input from the GUI and allows data collection prior to the data being put on the database. The subsystem is programmed in Python using both the Google Cloud Platform and the Microsoft Graph Platform to implement Google Calendar and Outlook APIs. Both Calendar APIs allow a user to verify permission for the Smart Scheduler to access their data before our program retrieves and organizes it properly before sending the necessary information to the database. The subsystem is also used to push finalized meetings onto user calendars and invite all attendees to them. Our initial design planned to have both Outlook and Google Calendar fully implemented by the end of the semester, but about halfway through, our sponsor changed the scope. Google Calendar is fully implemented, but our sponsor would like the app to undergo testing over the summer before we also implement the Outlook functionality. In the subsystem report below, the Google Calendar implementation will be discussed, followed by the research and Outlook code that will be integrated into the application either over the summer or during ECEN 404.

4.2. Google Calendar Pull

4.2.1. Operation

Google Calendar Pull (GCal Pull) takes the inputs: deadline and user, where the deadline holds all of the known information about the meeting and user is the contact address of the user providing their calendar. These inputs are received from the GUI. GCal Pull begins by taking a user out of the GUI and sending them to an OAuth2.0 (Google) authentication page, shown below, where they are informed about the permissions that the subsystem is asking for. When allowed, it generates a token under the user's contact address and uses it to extract every event that the user has on their calendar between the time it was accessed and the last possible meeting day. It then records each event's start and end time as well as the number of attendees involved with each meeting by creating a data structure with the aforementioned information. GCal Pull completes by outputting the data structure, the contact address of the user, and the name of the meeting to be scheduled to the database subsystem before deleting the token that was generated to ensure that a user must re-authenticate and grant the subsystem permission every time that it is used. This is done mostly for security reasons, due to the low-security efforts

of the system as a whole. Deleting tokens also helps with organization by associating pulled data with the user that it was pulled from.

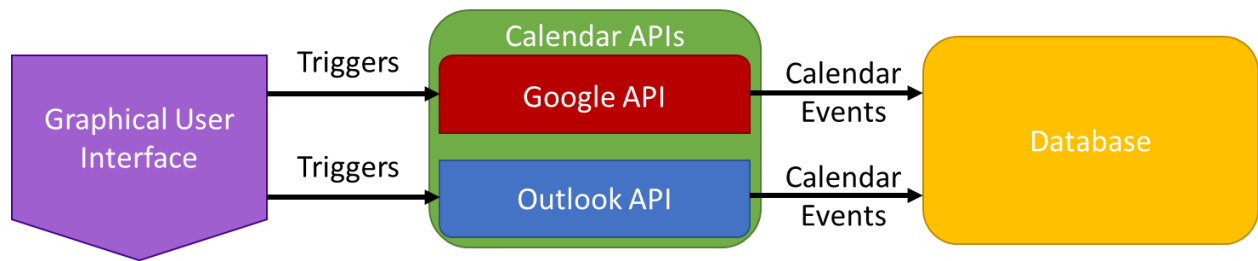


Figure 17: API Subsystem Block Diagram

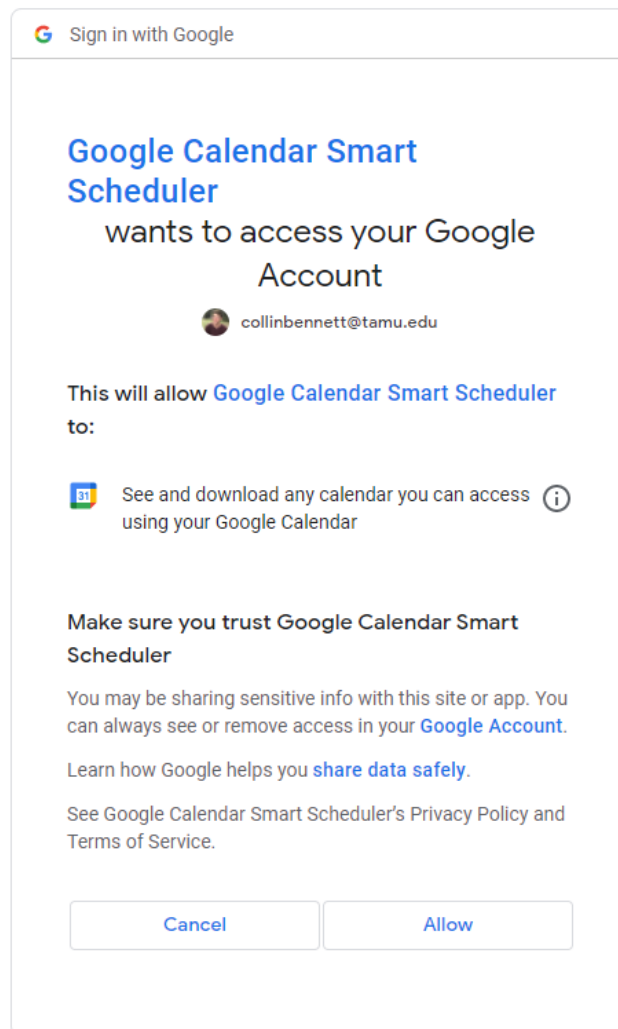


Figure 18: OAuth2.0 request page

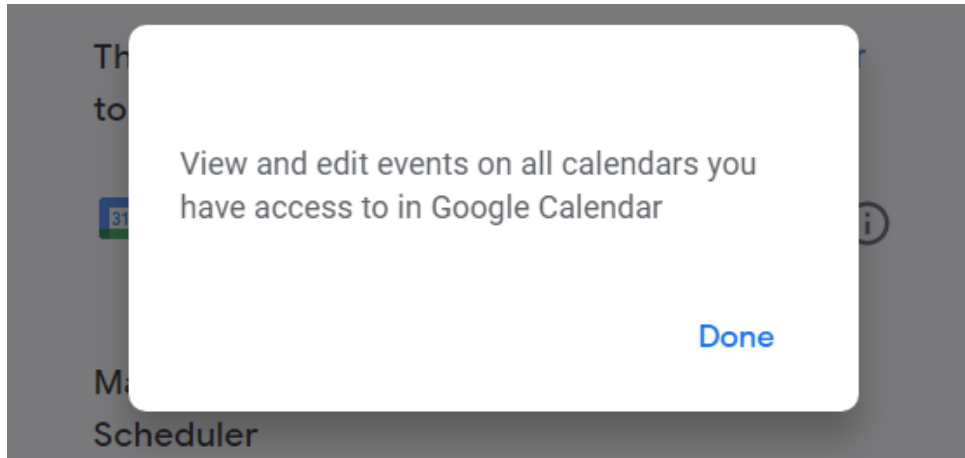


Figure 19: OAuth2.0 permissions

```
# API call
events_result = service.events().list(calendarId='primary', timeMin=now,
                                     timeMax=later, singleEvents=True,
                                     orderBy='startTime').execute()
```

Figure 20: API Call code snippet

```
# Prints the start and name of the events from today until the deadline
for event in events:
    # start holds a given event's start time
    start = event['start'].get('dateTime', event['start'].get('date'))
    # end holds a given event's end time
    end = event['end'].get('dateTime', event['end'].get('date'))
    try:
        # if an event holds values for attendees, att holds them
        att=len(event['attendees'])
    except:
        # if there is no attendees object in the event, set att to 1 (the user)
        att=1
```

Figure 21: Data selection and storage code snippet

4.2.2. Validation

Google Calendar Pull was first tested by ensuring that all user meetings between the time accessed and a test date were properly pulled from a user's Google Calendar and displayed properly onto the terminal. When operational, it displays event start and end times as datetime values containing the date and the time for each value, as well as the number of attendees associated with each meeting, and displays 1 when no attendees are found. The subsystem was then tested by connecting it to the database subsystem and ensuring that all of the information it received was passed into the database in an organized manner. More on this in the database subsystem.

4.3. *Google Calendar Push*

4.3.1. **Operation**

Google Calendar Push (GCal Push) is used when a meeting date has been chosen to add the meeting to the host's calendar. This is done by creating a meeting in an applicable format to Google Calendar by providing a start time, end time, as well as a summary, description, locations, and list of attendees, this is shown below. Once this has been scheduled on the host's calendar, the meeting gets automatically shared with all of the attendees listed through Google Calendar's systems without any extra input.

```
# the event to be pushed onto google calendar
event_result = service.events().insert(calendarId='primary',
    # the contents of the event, in json format
    body={
        "summary": summary,
        'location': location,
        "description": description,
        "start": {"dateTime": e_start, "timeZone": "America/Chicago"},
        "end": {"dateTime": e_end, "timeZone": "America/Chicago"},
        'attendees': att,
    }
).execute()
```

Figure 22: GCal Push code snippet

4.3.2. **Validation**

Google Calendar Push was tested by creating a meeting and posting it onto a user's Google Calendar. Once this was successfully completed, a meeting with a list of attendees was created and then pushed to check for a successful push on all attendees' calendars. Both of these tests were completed and validated.

4.4. *Outlook Calendar - Pull Events*

4.4.1. **Operation**

When pulling events from a user calendar, the specified user email, start time, and end time must all be specified. For use with the Smart Scheduler, the current time is used as the start time, and the end time is determined by the deadline entered by the meeting creator. Once the Outlook calendar request is triggered, the user is redirected to an Outlook authentication screen that is very similar to the Google OAuth2.0 interface. Outlook's authentication window informs the user of the permissions that must be given for sharing calendar data. When allowed, it generates a token tied to the user's Outlook email and uses it to extract every event that the user has on their calendar between the start time and the end time.

collinbennett@tamu.edu

Approval required

unverified

This app requires your admin's approval to:

- ✓ Read and write user and shared calendars
- ✓ Read user and shared calendars
- ✓ Sign in and read user profile
- ✓ Read user calendars
- ✓ Have full access to user calendars

Enter justification for requesting this app

[Sign in with another account](#)

Does this app look suspicious? [Report it here](#)

Cancel

Request approval

Figure 23: Outlook permissions request menu

```
# define the protocol being used and the scopes of our app
protocol = MSGraphProtocol()
#protocol = MSGraphProtocol(default_resource='<sharedcalendar@domain.com>')
scopes = 'https://graph.microsoft.com/.default'
account = Account(credentials, protocol=protocol) #, tenant_id = TENANT_URL

if account.authenticate(scopes=scopes):
    print('Authenticated!')

# create an instance of the schedule
schedule = account.schedule()
# create an instance of the calendar
calendar = schedule.get_default_calendar()
# starting a new calendar query for the events in the user calendar
q = calendar.new_query('start').greater_equal(dt.datetime(2019, 11, 20))
q.chain('and').on_attribute('end').less_equal(dt.datetime(2023, 11, 24))
#events = calendar.get_events(include_recurring=False)
events = calendar.get_events(query=q, include_recurring=True)
# for loop to print all the calendar events
for event in events:
    print(event)
```

Figure 24: Outlook pulling all calendar events

4.4.2 Validation

Outlook Calendar's API pull feature was validated by creating an Outlook calendar and populating it with events. These events were pulled from the calendar using the function in the code above by specifying a start time and an end time. Once done, the pulled events were compared to the start and end times to see if all of the correct events were pulled. They were found to be retrieved correctly.

4.5. Outlook Calendar Push

4.5.1 Operation

Outlook's calendar API also allows users to insert events into specified user calendars using the function below. The function requires a starting datetime and an ending datetime. These will both be passed by the scheduler program. The event details required by the function include a subject (meeting name), content (meeting description), time zones, and a meeting location. Once all of these parameters are passed, the calendar event can be pushed to a specified user calendar. When this event is triggered, it prompts an OAuth2.0 redirect like the one seen when pulling events from an Outlook Calendar.

```
from datetime import datetime, timedelta

start_datetime = datetime.now() + timedelta(days=1) # tomorrow
end_datetime = datetime.now() + timedelta(days=1, hours=1) # tomorrow + one hour
timezone = "America/Bogota"

data = {
    "calendar_id": "CALENDAR_ID",
    "subject": "Let's go for lunch",
    "content": "Does noon work for you?",
    "content_type": "text",
    "start_datetime": start_datetime,
    "start_timezone": timezone,
    "end_datetime": end_datetime,
    "end_timezone": timezone,
    "location": "Harry's Bar",
}

response = client.calendar.create_event(**data)
```

Figure 25: Outlook pushing a calendar event

4.5.2 Validation

Outlook Calendar's API event push feature was validated by pushing events with differing parameters to a test calendar and ensuring that each created event was created at the right time and displays the right information. Only the research for Outlook's API has been conducted, as well as programming the ability to push and pull calendar events. In ECEN 403, these functions will be implemented into our program alongside the already-implemented Google Calendar functionality.

4.6. Subsystem Conclusion

The subsystem was not only tested and functional as a standalone system, but is working properly when connected to the GUI and database subsystems, adequately serving its purpose as the in-between. The subsystem properly constructs a list of user event data for every event and is able to pass this data in a readable way into the database for it to store.

5. Database Subsystem Report

5.1. Subsystem Introduction

The Database subsystem is how information gets stored, accessed, and deleted from, by, and for all of the other subsystems. The Database subsystem is designed to take inputs from the GUI and Calendar subsystems and store provided information to later be outputted back to the GUI, to the Search Algorithm subsystem, and to the Email notification subsystem. This subsystem is also used for semi-permanently storing data from the results of the Search Algorithm subsystem in an effort to improve upon the design of the Smart Scheduler in its next version. All database operations are done through the Smart Scheduler firebase real-time database, chosen for its familiarity and ease of use.

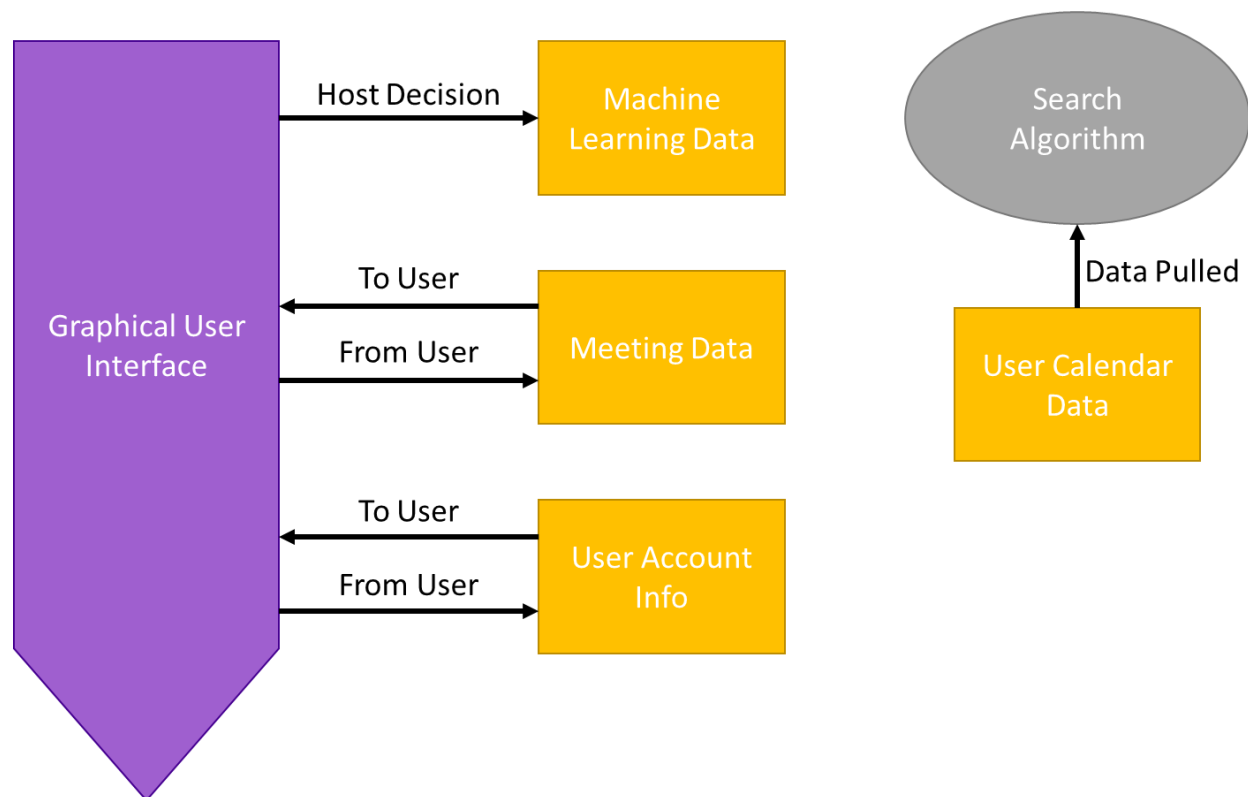


Figure 26: Database Subsystem Block Diagram

5.2. *Firestore Push*

5.2.1. Operation

Firestore Push is a method used by the Calendar API subsystems to take user event data from the Google Calendar subsystem and place it into the database. The method of sorting the event data is shown below. The information is stored under Meetings>[meeting name]>[user contact address]>[random pid]>start,end,summary in .json format. This allows us to hold all meeting data separate from the user data and ML data while also separating the data by which meeting it is associated with as well as who's data it is. This is useful in the Search Algorithm subsystem to denote who has a conflict at a given time for what meeting.

5.2.2. Validation

Validation for Firestore Push was done by giving the function dummy data and ensuring that it was capable of accessing the database for the Smart Scheduler. Once it was able to achieve this, validation then moved up to receiving real data from the Calendar API subsystem and placing it in an organized manner under the Meetings directory. The tests for Firestore Pull were completed successfully.

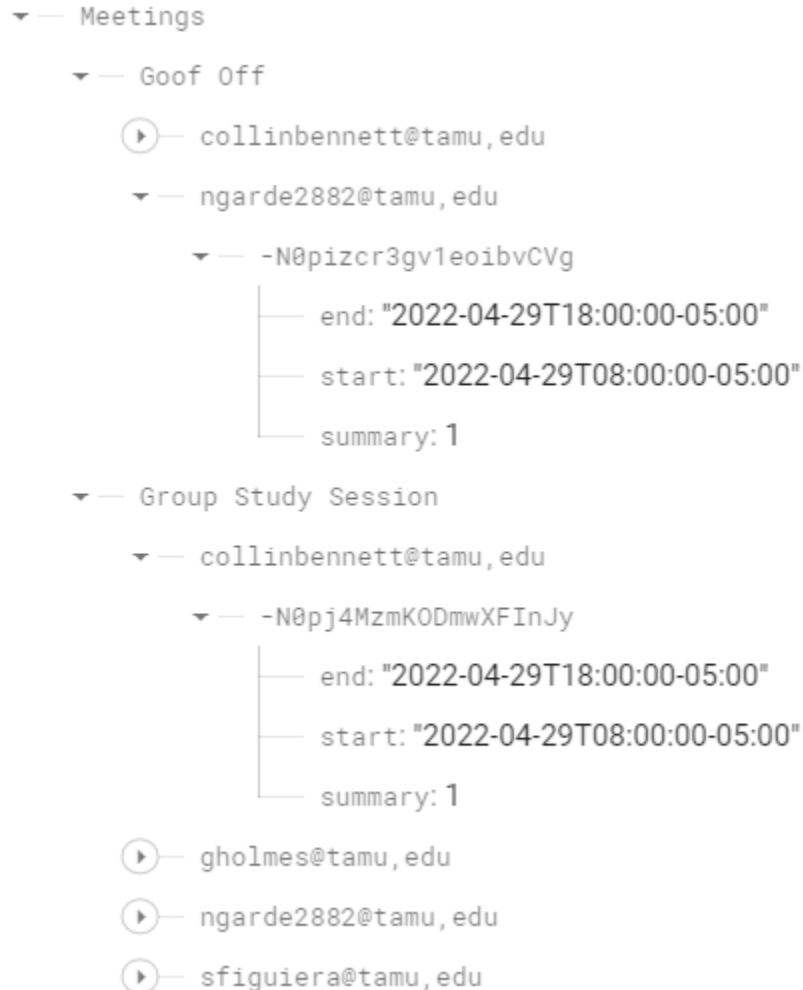


Figure 27: Firestore Push storing user events onto the database

5.3. *Firestore Pull*

5.3.1. Operation

Firestore Pull is a method of retrieving selected data from the database and passing it to the Search Algorithm subsystem, deleting it from the database once this is completed. This data then needs to be reconfigured back into lists and classes from the .json format received which is done before outputting to the Search Algorithm subsystem. Typically, once all users have provided their schedule data and it is stored in the database, the meeting host will then attempt to find good meeting times for their meeting. To achieve this, the data that was previously stored in the database needs to be reached, which is done by Firestore Pull.

5.3.2. Validation

Firestore Pull was first validated by ensuring that it could output data found in the database onto the terminal. Once this test was successful, the next step was proving that it could pass this data in a readable way into the Search Algorithm subsystem. Finally, once both of these things were completed, the last test was to ensure that the subsystem was deleting the retrieved data out of the database to maintain the cleanliness of the database. Firestore Pull passed all validation tests.

5.4. *ML Push*

5.4.1. Operation

ML Push is a method that retrieves data from the Search Algorithm subsystem about the options that were provided to a host for scheduling, and stores the options as well as the selected option in the database for later use. To do so, this data must be translated into a .json format for the database to accept it. Next semester will begin the work on version 2 of the Smart Scheduler and collecting data from what users preferred will help guide the next iteration to become even more successful at suggesting meetings that users will want.

5.4.2. Validation

The validation for ML Push began with giving it dummy data and ensuring that it could properly store it in the database, once this was achieved the final validation was testing if it could retrieve this data from the Search Algorithm subsystem to place onto the database. ML Push was completely successful in these tests.

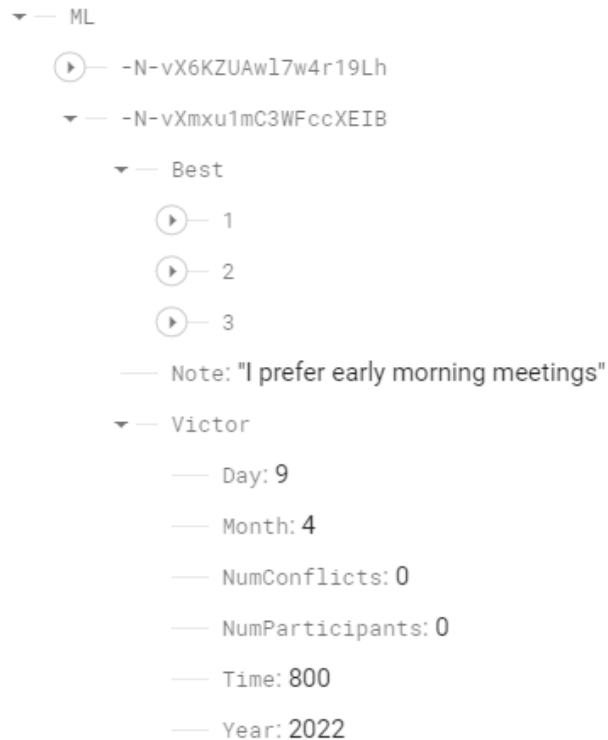


Figure 28: ML Push data stored in the database

5.5. Account Data

5.5.1. Operation

User account data is stored in Firebase. This data includes user credentials, an email list, and stored meeting information. When a user initially signs up to use the Smart Scheduler, they insert a first name, last name, email, and password. The password is stored using Google Authentication, but all other information is stored under user credentials, along with a personal user identification code.



Figure 29: User credentials stored in the database

A personal user email list is also stored in Firebase. This is strictly for convenience purposes and ensures that the user will not have to enter a list of email addresses every time they wish to create a meeting. Before creating a meeting, users have the option to create meeting contacts

by entering a name and contact email address. This data is stored in Firebase. Once a contact is stored in this list, the user can simply select the corresponding contact name and add it to an attendees parameter tab on the meeting creation screen in the GUI.

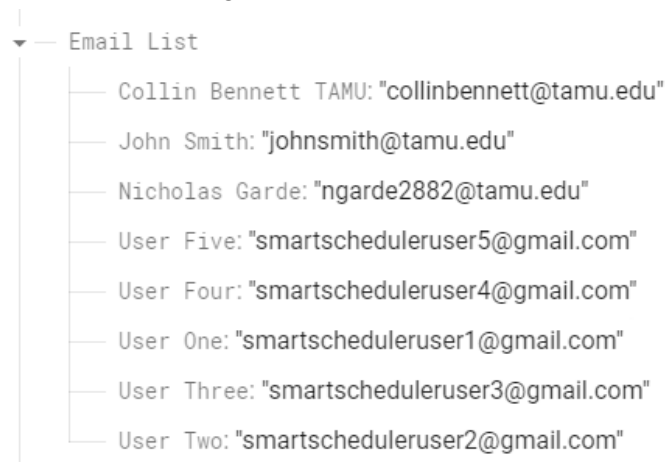


Figure 30: User email list stored in the database

Meetings and their associated information are also stored under users in Firebase. When a user schedules a meeting, the meeting name, attendee list, meeting deadline, meeting description, meeting duration, and meeting link/location are all added to Firebase. When attendees are added to the attendee list, they are placed under the category “Pending” by default. When users share their calendar data with the program or decline to, this status changes to “Attending” or “Not Attending”, respectively. When a user is attending, their associated Google Calendar email address is added to a list in firebase so events can be pushed to each of their calendars. Once events are pushed to calendars, the scheduled meeting is deleted from the user’s database, protecting confidentiality.

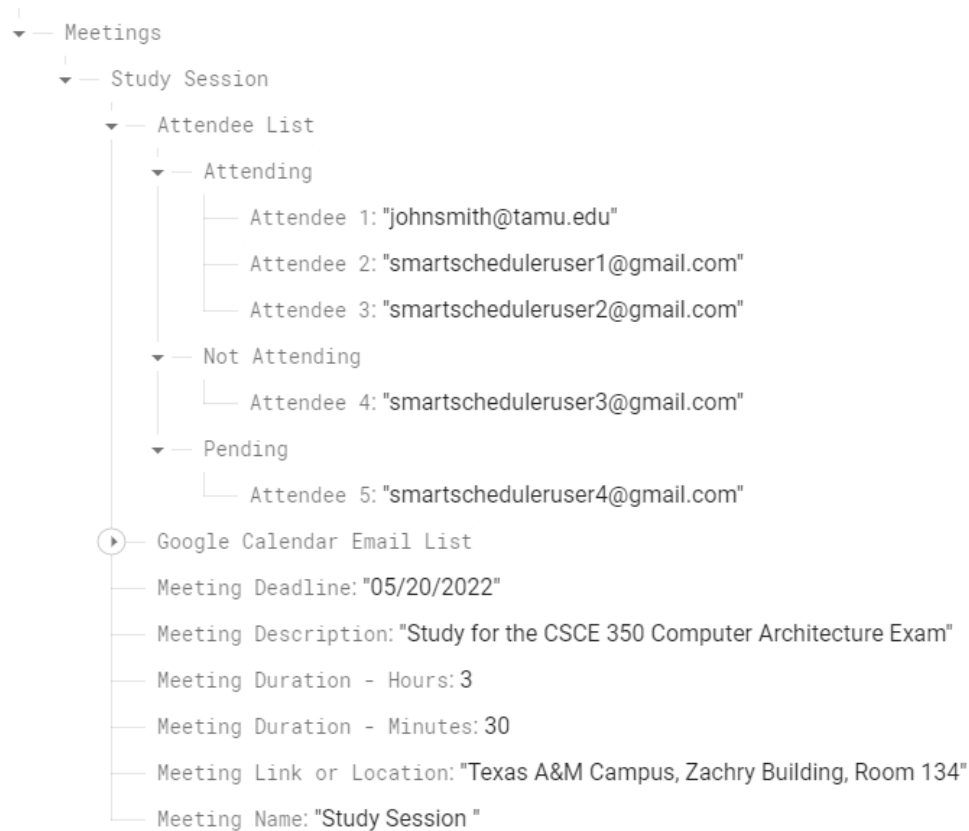


Figure 31: User meeting information stored in the database

5.5.2. Validation

Account data integrity and correctness were validated by inserting dummy data into the graphical user interface and ensuring that it was uploaded correctly to Firebase. This was done many times under different circumstances. Once this data was uploaded to Firebase, it was pulled for program use many times to ensure that the data is properly retrieved. The database is an integral part of the Smart Scheduler, supplying information to the GUI as well as emails, so it was extensively tested.

5.6. Subsystem Conclusion

All parts of the database subsystem were validated and working as intended. The subsystem is currently capable of uploading data to the database as well as retrieving and deleting it. The subsystem is behaving as intended when connected to the other subsystems as well. The database is a critical subsystem of the Smart Scheduler, and it needs to be able to cooperate with the other subsystems to properly function as a whole.

6. Email Subsystem Report

6.1. Subsystem Introduction

Integrated email notifications are vital to a program that automates meeting scheduling. Our program sends out emails for two different scenarios. In the first scenario, when an organizer creates a meeting, an email is sent out to invitees. This email contains meeting details and a link that potential attendees can follow to accept or decline the meeting. In the second scenario, when the scheduled meeting is pushed to attendee calendars, users are notified of the meeting. Our emails are sent using the Mailgun API and Simple Mail Transfer Protocol, an internet standard communication protocol for email transmission.

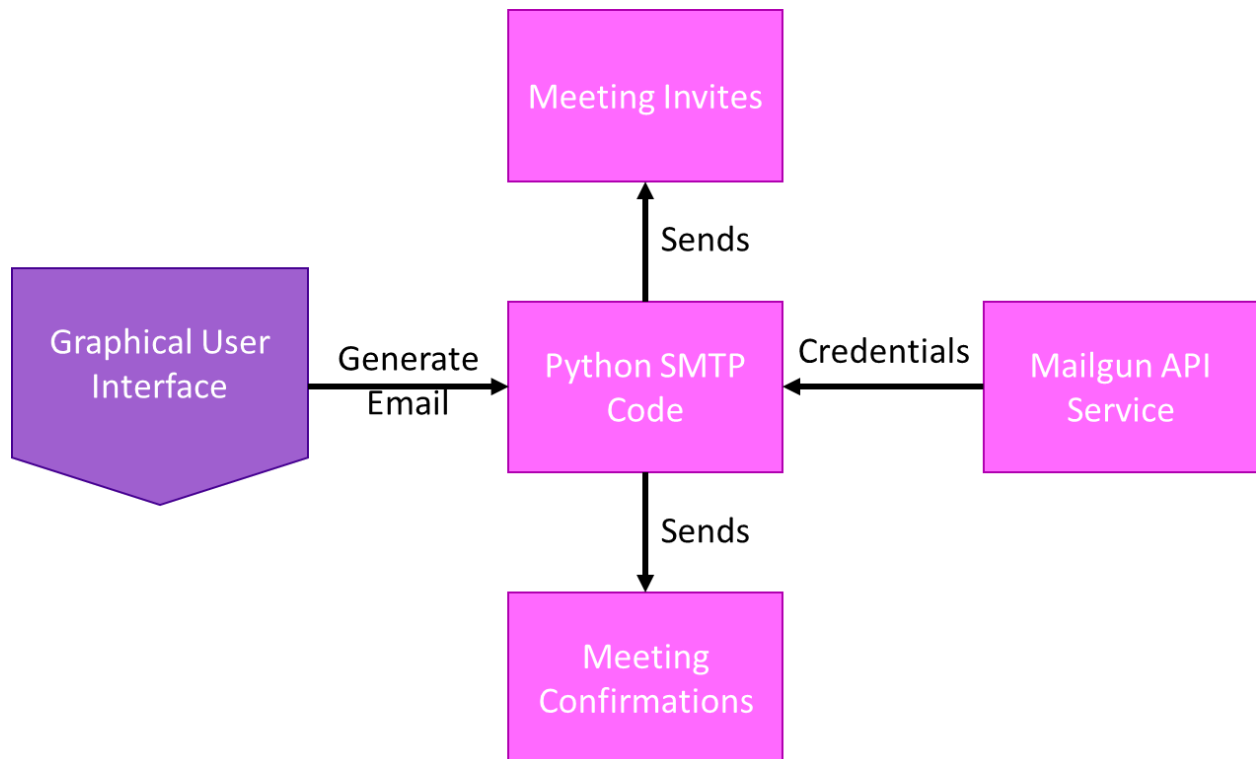


Figure 32: Email Subsystem Block Diagram

6.2. Mailgun and SMTP

6.2.1. Operation

Initially, emails were going to be sent using Sendgrid, but Mailgun was found to have a better user interface and improved reliability. A domain name was registered and set up to send and receive emails, and API keys were generated. Emails are best sent via SMTP relay when using mailgun, and an example function can be seen below in figure 33. In the code, meeting details are passed into the function as variables and the formatting of the email message is generated.

```
def email_invite(
    sender_name,
    sender_email,
    receiver_email,
    meeting_name,
    meeting_description,
    meeting_deadline,
    meeting_duration_hours,
    meeting_duration_minutes,
    meeting_link_location
):
    # the sender email is our custom domain email
    sender_email = "notifications@smart-scheduler.com"
    # API password - tighten security later

    message = MIMEMultipart("alternative")
    # subject line is simply the meeting name
    message["Subject"] = meeting_name
    # the from links to the sender email address
    message["From"] = sender_email
    # message["To"] = receiver_email
    password = ""

    # Create the plain-text and HTML version of your message
    # this message below includes a link the the RSVP page for the scheduling application
    text = f"""
    You have been invited to a meeting hosted by {sender_name}.
    Here are the meeting details:
    Meeting Name: {meeting_name}
    Meeting Description: {meeting_description}
    Meeting Deadline: {meeting_deadline}
    Meeting Duration: {meeting_duration_hours} hours and {meeting_duration_minutes} minutes
    Meeting Link/Location: {meeting_link_location}

    To RSVP for this meeting, go to https://user-form-2022.herokuapp.com/
    """
```

Figure 33: Email code, body message

```

# Turn these into plain/html MIMEText objects
part1 = MIMEText(text, "plain")
part2 = MIMEText(html, "html")

# Add HTML/plain-text parts to MIMEMultipart message
# The email client will try to render the last part first
message.attach(part1)
#message.attach(part2)

# need to pass in correct meeting information here
try:
    smtpObj = smtplib.SMTP('smtp.mailgun.org', 587)
    smtpObj.starttls()
    smtpObj.login(sender_email,password)
    print("Login successful")
    smtpObj.sendmail(sender_email, receiver_email, message.as_string())
    print("Email sent!")
    print(message)

# exception handler in case the email fails to send
except Exception:
    print("Error: unable to send email")

```

Figure 34: Email code, SMTP relay

In the code above, the SMTP object is generated and sent using the Mailgun service on port 587. Transport Layer Security (TLS) is used to encrypt the email, ensuring that it will not be intercepted and eliminating any privacy concerns. If the email is sent incorrectly, an exception is thrown, and this can be referenced later for any possible debugging.

6.3. *Meeting Invitation Email*

6.3.1. Operation

When a user creates a potential meeting using the GUI, a meeting invitation email is sent out to all potential attendees on the invitation list. This email contains all of the information necessary for an attendee to make an informed decision on accepting (or declining) a meeting. The meeting name is included in the subject line, as well as the body of the text. The meeting description, duration, and deadline are also included. The meeting link/location is also included, and in the future, the Google Maps API will be integrated so users can see how far away a meeting is. The final portion of the email contains a link redirecting to the portion of the Smart Scheduler application that allows users to accept or decline meetings. Currently, the web app is being developed locally, and hosting is something that will be implemented at the beginning of the summer for beta testing of the scheduler

Demo Meeting Inbox x



notifications@smart-scheduler.com

to ▼

You have been invited to a meeting hosted by Collin Bennett.

Here are the meeting details:

Meeting Name: Demo Meeting

Meeting Description: Meeting for 403 Demo

Meeting Deadline: 2022-05-03

Meeting Duration: 1 hours and 0 minutes

Meeting Link/Location: FEDC

To RSVP for this meeting, go to <https://user-form-2022.herokuapp.com/>

Figure 35: Meeting invitation email example

6.3.2. Validation

The invitation emails were validated by ensuring they sent the correct meeting information to the correct user successfully. Currently, emails are successfully being sent to users, but there is a slight chance that they are being sent to spam folders. This can be attributed to the relative newness of the domain, notifications@smart-scheduler.com. Over time, as more emails are sent and the domain matures, these emails will no longer be sent to spam folders. HTML/CSS styling will also be implemented into the emails, giving them a more professional appearance, which will also decrease the likelihood of the emails being flagged as spam. In regards to the content of the email, many were sent out with varying contents, and all were found to contain the correct information formatted in the correct way.

6.4. *Meeting Confirmation Email*

6.4.1. Operation

After a meeting creator has pushed meeting events to all attendee calendars, a meeting confirmation email is sent to all attending members, informing them of this calendar insertion. This email contains the meeting name, creator, time, and link/location. No action is required from the meeting attendees, but a general google calendar link has been added to the email, so users can see the event in their calendars.

Demo Meeting Confirmation Inbox x



notifications@smart-scheduler.com

to ▼

A meeting has been added to your Google calendar by Collin Bennett.

You may choose to accept or decline this meeting.

Here are the meeting details:

Meeting Name: Demo Meeting

Meeting Time: 12:30 pm

Meeting Link/Location: FEDC

<https://calendar.google.com>

Figure 36: Meeting confirmation email example

6.4.2. Validation

The validation for confirmation emails is the same as that for the meeting invitation emails. Many emails were sent out and checked for accuracy and content correctness. Changes will be made to the emails in the future to reduce the slight possibility of getting flagged as spam.

6.5. *Subsystem Conclusion*

All parts of the email subsystem have been validated and are working as expected. The subsystem is currently capable of sending out meeting invitation emails and meeting confirmation emails. The email subsystem is integral to the Smart Scheduler program, as without it, users would be unable to accept or decline meetings. In ECEN 403, styling will be implemented in an effort to eliminate emails being flagged as spam.