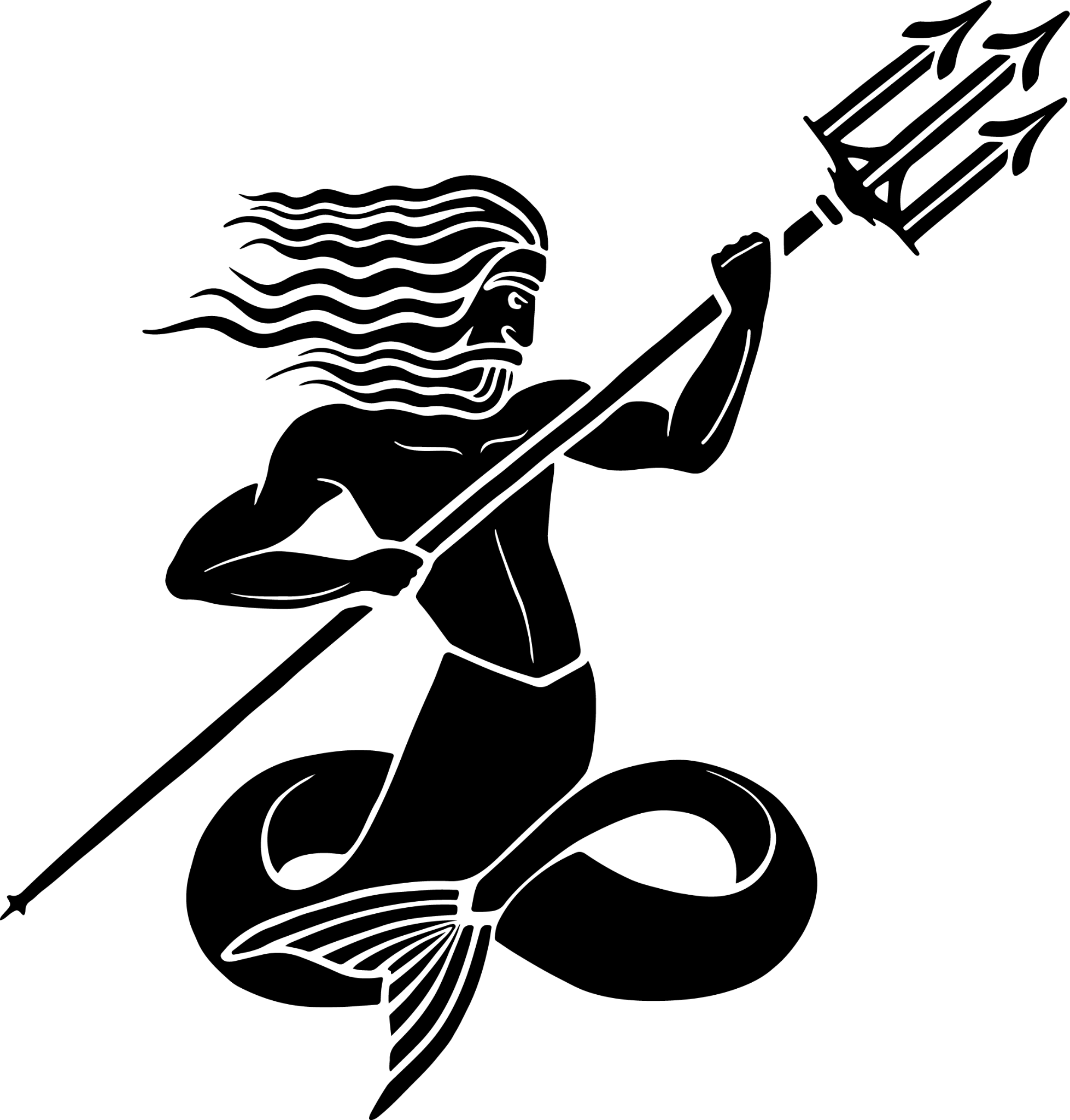
AGILE v3.0

**A**dvertising **G**eolocation **I**nformation **L**ogical **E**xtractor

**Manual**



MIDN 1/C Sirsikar, Summers, and Traynor (CS)

Team Lead: MIDN 1/C Summers

## Customer: **Landon Haggard**, NSWDG, landon.d.haggard.mil@socom.mil, 832-465-4832

## 

## 

## Table of Contents

System Introduction 3

Purpose of the Manual 3

General Overview 3

System Setup (Linux) 4

Cloning the Repository 4

Installing Dependencies 4-5

Installing and Running (w/ Docker) 5

Running (w/o Docker) 5

Notes on Setup and Running 5

Important Folders and File Structure 6

High-Level View of AGILE 7

User Start Tutorial 8-10

Generating a Graph for an ADID Dataset (Houthi Example) 11-12

Dictionary of AGILE’s Most Useful Features 13

Detailed View of System 14-16

Overview of graphing.py 18-22

App.py Explained 23-28

Where to Go From Here 29-31

Appendix 32

Quick Reference Card 32

System Requirements 33

Installation 33

Troubleshooting/Known Bugs 34

Maintenance Procedures and Issues 34

Developers’ Information 35

## 

## System Introduction

### 

### **Purpose of the Manual**

AGILE is a legacy project which has been worked on by midshipmen for the past three years. For future groups to contribute effectively, this manual provides developers all of the necessary information to get up to speed on the project. The manual begins with a general overview of the project, but then it dives into the specific code and functions, explaining what each does. Finally, the manual concludes with recommendations for next step action items as a starting place for future groups. While this manual is a continuation of previous groups’ work, it focuses heavily on the additions and progress made by the Class of 2025. Overall, this manual is a dictionary that enables future groups to download and run the website, learn the ropes, and then be able to build upon it quickly and efficiently. One of the hardest parts with working on a legacy project is attempting to understand the previously written code. It is our hope that this manual aids you in understanding what the previous groups have contributed, and that you us it to look up anything you come across that you do not understand. We have also attached appendices with further information that we believe endpoint users will value highly. Please let us know if you have any questions and we hope you enjoy perusing the AGILE Manual.

### 

### **General Overview**

Our project’s goal is to continue work on the last several years’ AGILE project in order to make it easier for the customer, NSWDG, to make sense of Advertising Identification (ADID) data. The customer had identified that an issue with AGILE was that it returned a high volume of data which the user had to make sense of manually through various functions and tabs. We streamlined this process by creating a new graphing function. We hope this visualization and automation will help the customer answer complex questions and identify links between potential targets. To interact with the graph, users will enter ADIDs into the AGILE User Interface. AGILE will then do the work required to generate a graph, display the graph, and then enable the user to explore links between nodes.

## System Setup (Linux)

1. **Cloning the Repository**

Open a terminal and clone the repository into a folder using a GitHub or GitLab account. For Linux users, the command is:

git clone https://github.com/atrayn1/AGILE-Share-AY25.git

or

git clone git@github.com:atrayn1/AGILE-Share-AY25.git

The repository can be accessed at <https://github.com/atrayn1/AGILE-Share-AY25>.

1. **Installing Dependencies**

A number of dependencies need to be installed before running AGILE. This can be done by:

Using solely pip:

pip install streamlit==1.20.0 proximitypyhash==0.2.1 pygeohash==1.2.0 pandas==1.5.2 matplotlib==3.6.2 Bokeh==3.0.2 streamlit-folium==0.7.0 geopy==2.3.0 scikit-learn==1.2.1 fpdf2==2.6.1 networkx==3.0 streamlit-option-menu==0.3.

Using conda with pip:

conda install -c conda-forge proximitypyhash=0.2.1 pandas=1.5.2 matplotlib=3.6.2 bokeh=3.0.2 streamlit-folium=0.7.0 geopy=2.3.0 scikit-learn=1.2.1 fpdf2=2.6.1 networkx=3.0

pip install pygeohash==1.2.0 streamlit-option-menu==0.3.2

Using mamba with pip: The same process as above, but replace *conda* with *mamba*

If one method does not work, try another method. For installation of conda or mamba, see:

* Conda: <https://docs.anaconda.com/free/anaconda/install/index.html>
* Mamba: <https://github.com/conda-forge/miniforge>

The dependencies are also contained within AGILE/requirements.txt for use with the docker image.

You can also create a conda or mamba environment using agile.yml in the AGILE directory, by running:

mamba env create -f agile.yml

1. **Installing and Running Using Docker (skip if not using Docker)**

Navigate to the AGILE directory. Run these commands for the recommended default installation and running of AGILE using Docker:

foo@bar:../AGILE$ source /bin/build.sh

foo@bar:../AGILE$ build # To build the docker image

foo@bar:../AGILE$ run # To run the docker image

The manual installation and run commands are:

foo@bar:../AGILE$ source /bin/build.sh

foo@bar:../AGILE$ build # To build the docker image

foo@bar:../AGILE$ run # To run the docker image

NOTE: You may have to run either set of install commands with root permissions if using a docker version <19.03 or running from WSL.

1. **Running (not using Docker)**

Navigate to the AGILE directory. Run the following command to run AGILE:

foo@bar:../AGILE$ streamlit run app.py

**Notes on Setup and Running**

* The file app.py is the main driver of the application. This contains the baseline code for the interface. The file is located within the AGILE directory, where you should be when running the application.
* If an error arises stating there is a missing directory, run the following to alleviate the issue. This issue occurs on some computers, but not all. Ensure you are in the AGILE directory

foo@bar:../AGILE$ mkdir saved\_data

**Important Folders and Files Structure**

*Italicized words are folders*

*AGILE*

├─ *agile*

│ ├─ *utils*

│ │ ├─ dataframes.py → Cleaning and preprocessing main dataframe

│ │ ├─ files.py → Loading in the *names* files and generating aliases

│ │ ├─ geocode.py → Reverse geocoding a geocode

│ │ └─ tag.py → Queries Overpass API to find nodes within a radius

│ ├─ centrality.py → Determines people/locations of interest

│ ├─ data\_reset.py → Functionality for resetting the data after altering it

│ ├─ filtering.py → Filtering the data by date, node, ADID, or location

│ ├─ graphing.py → Produces graphs that display relationships between ADIDs

│ ├─ locations.py → Finds locations of interest and repeated visits

│ ├─ mapping.py → Functionality for generating maps

│ ├─ overview.py → Finds value counts for the data overview

│ ├─ people.py → Determines co-location of people

│ ├─ prediction.py → Main file for ML/clustering algorithms

│ ├─ profile.py → Class for a profile of an ADID

│ └─ samsreport.py → Class for a report of a Profile

├─ *names*

│ ├─ first.txt → List of first names for aliases

│ └─ last.txt → List of last names for aliases

├─ app.py → Main driver for Streamlit interface/UI

│

├─ *data* (make this directory and put those files in it)

│ ├─ hainan.csv → Artificial data at a Chinese naval base

│ ├─ houthi\_adid\_dataset.csv → Artificial data of the Houthi example (demo)

│ ├─ test\_location\_data\_gh.csv → Artificial data from DEVGRU (w/ geohash)

│ └─ test\_location\_data\_no\_gh.csv → Artificial data from DEVGRU (no geohash)

├─ *tests*

│ └─*AY25*

│ ├─ test\_create\_graph.py → Tests graph creating functions

│ ├─ test\_csv.py → Tests whether or not csv file is parsable

│ ├─ test\_frequency.py → Tests findFrequencyOfColocation() function

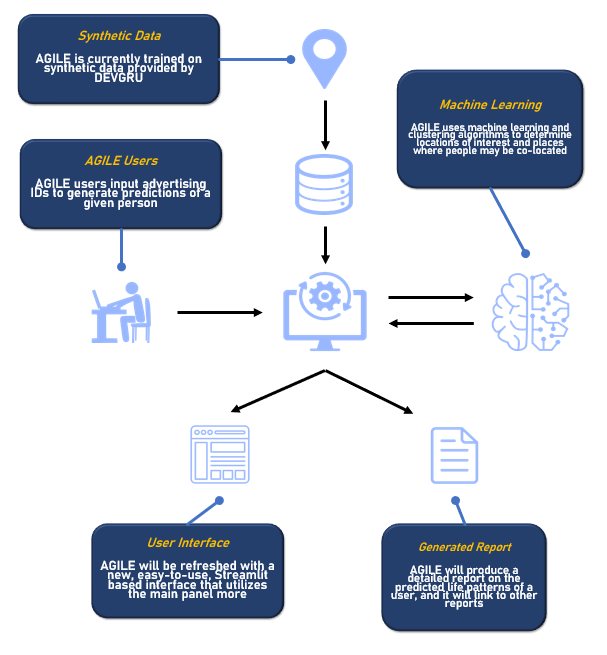
│ ├─ test\_hainan.py → Tests the graph generated by hainan dataset

│ ├─ test\_merge.py → Tests merging of collocation and dwell time

│ └─ testgraph.py → Has function needed for importing data

└─ visual\_graph.py → Creates a visual representation of the adjacency matrix

## High-Level View of AGILE



## User Tutorial

## 

**Upload Your Data**

## The first step in using AGILE is to upload the data you would like to utilize. Data must be in a specific format, shown in the second picture below.

## 

## To upload data, first ensure you are in the ‘Data’ portion of AGILE, with the navigation bar displayed on the left hand side of the website.

## 

## 

## Next, click the ‘Upload Data’ button, select your data, and wait for it to upload and geohash. This will be done when the ‘RUNNING’ icon in the top right corner disappears.

## 

**Filtering Your Data by ADID**

## Often, datasets contain extraneous information that slow the process of using AGILE. Filtering allows users to remove unnecessary data by ADID, Location, and Time.

## 

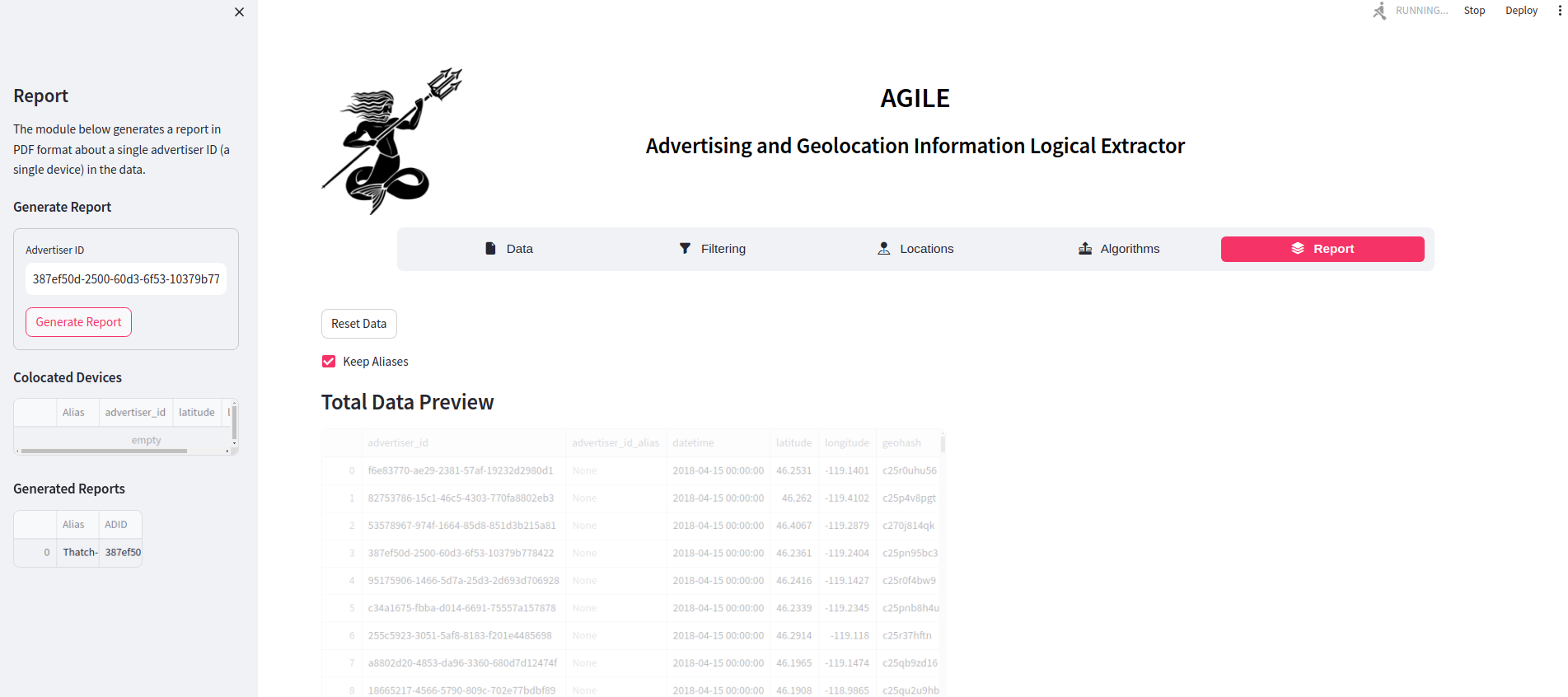
## To filter by ADID, first navigate to the ‘Filtering’ portion of the website. Then, identify an ADID that you are interested in. Then, put that ADID into the ‘Advertiser ID’ portion of the website and click ‘Query.’ This will be done when the ‘RUNNING’ icon in the top right corner disappears.

## 

**Generating Reports on an ADID**

## The best way to gain information on most ADIDs will be the reporting feature. To generate a report on an ADID, navigate to the ‘Report’ portion of the website, enter your interested ADID into the ‘Advertiser ID’ portion of the website, and click ‘Generate Report.’

## 



## This will be done when the ‘RUNNING’ icon in the top right corner disappears. Keep in mind that when data has been filtered, the Report Generation feature will only use the filtered data.

## This is an example of a report with minimal information and of the header format of our reports as it appears on the website.

## 

**Generating a Graph for an ADID Dataset (Houthi Example)**

## [Refer to our [slideshow](https://docs.google.com/presentation/d/1CE-v7UUhF05vEcetb2hWRHVt5j41KMEiII1KCOuFp8w/edit?usp=sharing) for more information]

## The best way to explore and visualize connections between ADIDs will be by using the graphing feature. In order to provide you with a better understanding of the graphing feature and some of the other features in the website, this section will walk you through an example using the artificial Houthi data we created. Please keep in mind that this dataset is extremely small in scale.

## 

## As you watch the [video](https://drive.google.com/file/d/1aVkBpLXYFPCgEsx7AKq3wJU44Sr_Rz0i/view?t=19) provided, follow along by reading the following section, which explains what is going on in the video:

## 

The first step involves loading in a dataset (you can also press the ‘Load Demo Data’ button, and provided you have the ‘houthi\_adid\_dataset.csv’ file saved in the ‘data’ directory, that will also work). After loading in the dataset, you can quickly look over the data and see how often each AdID occurs in the dataset, time distribution, and geohash distribution (which tells you which locations are most popular within the dataset)

The next step involves navigating to the ‘Graph’ portion of the website. First you must generate a graph. You can do so over the whole dataset or around a particular AdID. If you simply just hit the ‘Generate Graph’ button, you will see a graph that displays the important groups in the dataset. You will be able to see the different groups that the dataset contains.

Now, say there is a particular AdID that interests you like adid\_1. You can build a graph around that particular AdID. In the example, I reduced the number of visible nodes to at most 5. Upon generating that graph, you can see that adid\_1 is at the center, and the nodes that it is connected to are all around it. If you press the ‘Connect Current Nodes’ button, you can see how all the nodes that are currently displayed are connected with each other. Because we know that adid\_11 is also connected to other nodes, you can press on the ‘Expand adid\_11’ button to expand the graph to see those connections (this is the same as using the ‘Expand Current ADIDs’ function on the sidebar; pressing the button will also preload adid\_11 into the other parts of the website). The reason why the other AdIDs do not appear on the graph is because they are not connected to any other node in the dataset. So that would be adid\_4, adid\_5, etc… If you were to generate a graph around such a node, nothing would be displayed because there are no other nodes to connect it to.

Moving forward, the ‘Find groups for ADID’ function lists all of the groups that that particular AdID is a part of. A group must contain at least 3 members and each member of the group must have a connection to every other member in the group. The order of the returned list does not signify anything about the groups.

Next, if you go to the ‘Filtering’ tab, you can see on a map where exactly adid\_1’s location pings appear. There will be way more pings for larger datasets. But you can see that adid\_1 has 2 pings in Yemen. One of them being at the Al Gishla Castle and the other being a building across the street from the castle. Then we can see where adid\_2’s pings occur. Notice that adid\_1 and adid\_2 have pings in 2 of the same locations. We can now see where adid\_11’s pings occur, because adid\_1 and adid\_2 and adid\_11 all form a group. Upon doing so you see that adid\_11 also has a ping at the Al Gishla Castle, along with adid\_1 and adid\_2. So we have a good idea that the castle is a meeting spot for the three people.

If you move over to the ‘Locations’ tab, you can use the ‘Overpass Polyline Query’ function to see places of interest along the path of a certain AdID. If you search for a 50 meter radius around adid\_11, you will see that the castle pops up again, this time as a location of interest. Move over to the ‘Algorithms’ tab and you can query the top n clusters for a certain AdID. If you do it for adid\_11, you will see the same locations pop up, but doing so enables you to be able to use the ‘Colocation’ function. The ‘Colocation’ function adds a blue mark to a cluster location if there are other AdIDs that show up at the location within a given time frame. In the video you can see that the search time is 2 hours. That may be a little long, so feel free to reduce it to 1 hour. But you can see that adid\_11 is collocated with adid\_1 and adid\_2 within the time frame at the castle, which strengthens our claim that those 3 people are connected because they meet at that castle. If the castle has been identified as a Houthi hotspot, then you can make a hypothesis that those individuals are associated with the Houthi organization. At the very least you can set up surveillance at that location. The ‘Movement Prediction’ function is the next most useful tool because it predicts where a certain AdID will be at a given time on a given day. In the video, only adid\_11 is run through the ‘Movement Prediction’ function, but you would want to run this on all the AdIDs of interest, so you have a better idea of when to set up surveillance or an airstrike. You can notice that it would be helpful to see the locations of all the AdIDs in the group at the same time. That would be a very helpful feature to add. Going back to the example, you can see that at that given time and day, adid\_11 will be at the castle. You can then assign adid\_11 with the nickname of ‘possible Houthi’.

Going back to the ‘Graph’ tab, if you have suspicions about adid\_11, you can generate a graph around that AdID. And to confirm, you can see that adid\_11, adid\_1, and adid\_2 form a group in the dataset.

**Dictionary of AGILE’s Most Useful Features**

* Data
  + Load Demo Data - you can either load a dataset or use the this button to load in the Houthi dataset
  + Add Alias for an Adid - assign a nickname to a specific AdID
* Graph
  + Initial Graph Creation - first step; create a graph to your specifications, either with all adids or a specific one you want to build around; you can also specify how many nodes you want displayed in the graph
  + Connect Displayed ADIDs - connects all the nodes that are currently displayed on the graph
  + Expand Current ADIDs - after the graph has been made, the clicking the buttons expand the graph around that specific AdID (Same as putting it into the Expand Current ADIDs); clicking the button, will also pre-load that AdID into the other forms throughout the website
  + Explore New ADIDs - expands the graph to include the specified AdID if it was not previously included in the graph
  + Find groups for ADID - returns a list of groups that the specified AdID is a part of (at least three members)
  + Filter Edges by Weight - provide limits for the edge weights
* Filtering
  + Advertising ID Filtering - shows you all the locations an AdID has been on a map
* Locations
  + Overpass Polyline Query - returns points of interest along the path of an AdID
* Algorithms
  + Top Clusters - to see top locations of interest for a single AdID
  + Colocation - returns the instances and location of devices that are collocated with the AdID that is clustered upon
  + Movement Prediction - predicts where an AdID will be given the day and time; good to use to make sense of what each location could be (work, house, etc…)

## 

## 

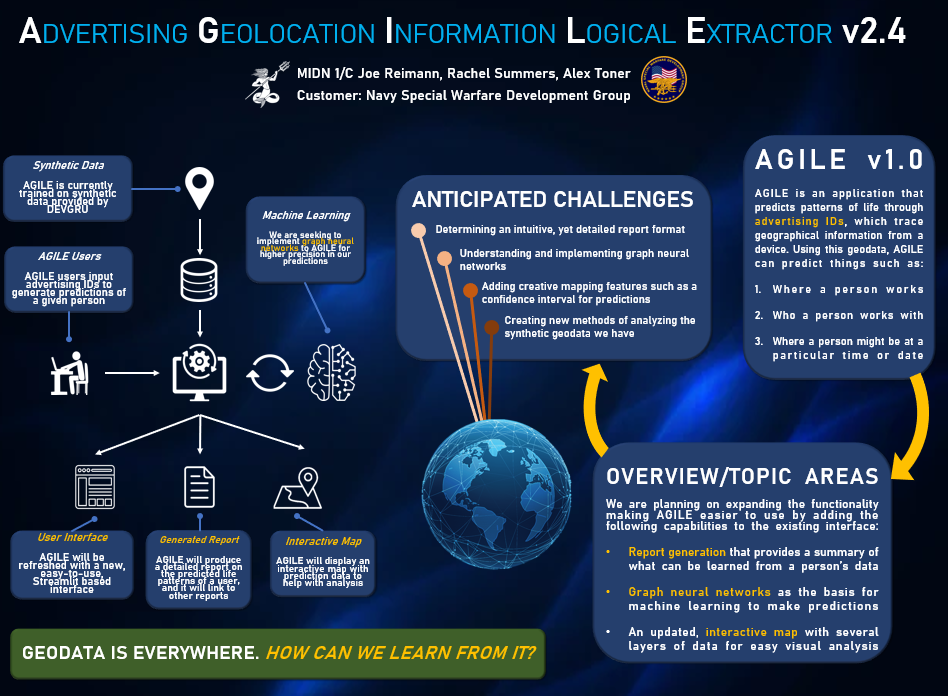
## Detailed View of System

## 

**Overall Goals of our System**

The goal of our system, AGILEv2.4, is to introduce report generation into AGILEv1.0 to provide our customer with a higher level of capability when analyzing geolocation data. This will allow them to input a target’s advertising ID and be able to gain significant amounts of useful information on them.

Our initial capstone poster is a useful tool to visualize this. It displays that our work is very much a continuation of and founded on AGILEv1.0. It displays our High Level Diagram, emphasizing the frequent interactions between users and the system, and the system and our data. It also displays potential challenges and focus areas for algorithms.



## 

## Narrowing down our focus, here is a high level view of the information stored in each report. They contain key locations, associated ADIDs, and other relevant information for each target. Reports are easily and intuitively generated using the ‘Generate Report’ button in the ‘Report’ section of the website.

## 

## 

## **Report Generation (Wireframe)**

## In order to generate the report, the AGILEv2.4 system does a significant amount of background work. This all begins when the user inputs an ADID. If the entered ID is invalid, they are prompted to enter a valid one. Otherwise, AGILEv2.4 runs multiple algorithms, links the ADID to others, utilizes aliases, and displays all relevant information to the user.

**Aliasing and Report Generation (Flow Chart)**

## 

## Once this report is generated, users have the ability to navigate between all generated reports on associated ADIDs, save the reports for future use, and use aliases to converse about ADIDs with other users.

## More specifically, each report is screened prior to display to determine AGILEv2.4’s confidence in the report it has generated. If the geolocation data provided does not include sufficient data on an ADID to ensure a confident report, AGILEv2.4 will display a warning message to users at the top of the report in order to allow them to have a proper level of trust in our system and make the best targeting decisions possible.

**Overview of graphing.py**

This script defines a graphing system that models relationships between nodes using edges. The nodes are represented as unique entities identified by an 'ADID'. These nodes can store a list of their neighboring nodes and the edges connecting them. Additionally, edges store information about the connection, including the weight, collocation count, overlap time, and the time periods during which nodes were collocated.

The graphing system allows for the creation, addition, and removal of nodes and edges. Each node can store multiple time periods, and these periods are used to calculate the temporal and spatial relationships between nodes. These relationships are used to generate weighted edges between nodes based on collocation frequency and overlap time.

**Node class**

The **Node** class represents a single entity in the graph, identified by a unique ADID. It stores various attributes such as original datapoints (latitudes, longitudes, timestamps), a list of neighboring nodes, edges, and continuous time periods.

The methods of the Node class include:

1. **getEdge(node)**: Checks if there is an existing edge between the current node and another node.
2. **add\_neighbor(node)**: Adds a neighboring node to the list of neighbors, ensuring uniqueness.
3. **remove\_neighbor(node)**: Removes a neighbor from the node's list.
4. **add\_edge(edge)**: Adds an edge connecting this node to another.
5. **remove\_edge(edge)**: Removes an edge connected to this node.

**Edge class**

The **Edge** class represents a connection between two nodes. It stores the nodes it connects, the weight (e.g., distance or strength), collocation count, overlap time, and the periods during which the two nodes were colocated.

Key methods of the Edge class include:

1. **forNode(node)**: Checks if this edge connects a particular node.
2. **addColocationCount(count)**: Adds collocation count to the edge.
3. **addOverlapTime(time)**: Adds the overlap time between the nodes.
4. **addOverlapPeriods(periods)**: Stores the overlap periods between the nodes.
5. **fixWeight()**: Computes the edge weight based on collocation count and overlap time.

**Graph class**

The **Graph** class manages a collection of nodes and edges. It supports operations such as adding and removing nodes and edges, managing an adjacency matrix, and storing additional matrices such as collocation and dwell time matrices.

Key methods of the Graph class include:

1. **get\_edge(node1, node2)**: Retrieves the edge between two nodes if it exists.
2. **add\_node(adid, features=None)**: Adds a new node with a unique ADID and optional features.
3. **remove\_node(node)**: Removes a node from the graph and updates the adjacency matrix.
4. **add\_edge(node1, node2)**: Creates an edge between two nodes.
5. **remove\_edge(node1, node2)**: Removes the edge between two nodes.
6. **get\_neighbors(node)**: Returns the list of neighboring nodes for a given node.
7. **get\_node\_by\_adid(adid)**: Finds a node by its ADID.
8. **get\_node\_names()**: Retrieves a list of ADIDs for all nodes.

**createGraph function**

The createGraph function initializes a graph from a dataset. It processes each data row to create nodes and assigns grid squares and time periods.

1. It calls preprocess to determine grid boundaries.
2. For each row in the data, it creates a new node or updates an existing one.
3. It assigns grid squares to nodes and associates each row's timestamp and coordinates with the node.
4. It stores continuous periods for each node based on time and distance.

**mergeResults function**

The mergeResults function combines two adjacency matrices. It element-wise divides the second matrix by the first to combine information, such as collocation and dwell time.

1. Converts the matrices into NumPy arrays for element-wise operations.
2. Checks if the matrices have the same dimensions.
3. Returns the merged matrix by performing the division.

**update\_graph\_with\_matrix function**

The update\_graph\_with\_matrix function updates the graph's adjacency matrix and neighbors based on the provided matrices.

1. Updates the adjacency matrix, collocation matrix, and dwell time matrix in the graph.
2. Iterates through all nodes and updates their neighbors based on the new matrix values.
3. Resets the neighbors and adds new ones based on the updated adjacency matrix.

**connectNodes function**

The connectNodes function connects nodes based on collocation frequency and proximity.

1. Calls findAllContinuousPeriods to compute periods where nodes were together.
2. Creates adjacency matrices for collocation frequency and dwell time.
3. Merges these matrices using the mergeResults function.
4. Updates the graph with new edges and adjacency matrix.

**findAllContinuousPeriods function**

The findAllContinuousPeriods function calculates continuous periods for each node, where consecutive data points meet the time and distance criteria.

1. Iterates through each node and calls get\_continuous\_periods.
2. Computes the continuous periods based on timestamp differences and distances between points.
3. Stores these periods for later graph operations.

**get\_continuous\_periods function**

The get\_continuous\_periods function identifies continuous periods where data points are close in time and space.

1. Iterates through the node's original data points.
2. Compares timestamps and distances using the haversine function.
3. Groups data points into continuous periods if they meet the criteria for time and distance.
4. Returns the periods with average coordinates.

**frequencyOfColocation function**

The frequencyOfColocation function calculates the number of times two nodes were colocated for a minimum time (x\_time) and within a specified distance (radius).

1. Uses a two-pointer approach to efficiently check overlaps in periods for both nodes.
2. Calculates the overlap duration between two periods.
3. If the overlap meets the criteria, it increments the collocation count.
4. Returns the number of colocations.

**findAllFrequencyOfColocation function**

The findAllFrequencyOfColocation function calculates a collocation matrix that stores the frequency of colocations between all pairs of nodes in the graph.

1. Iterates through each node pair and computes the collocation frequency using the frequencyOfCollocation function.
2. Returns the collocation matrix after populating it with the computed values.

**dwellTimeWithinProximity function**

The dwellTimeWithinProximity function calculates the overlap time between two nodes while considering their proximity.

1. Iterates through two nodes' periods and compares their timestamps and distances.  
   find\_cliques\_for\_adid function

**App.py Explained**

App.py is the main file where all the code for the main landing page of AGILE sits. This document serves as a developer guide for App.py. This application allows users to upload, filter, analyze, and generate reports from geolocation data associated with AdIDs.

**Overview**

The application is built using the Streamlit framework, providing an interactive web interface. It leverages various Python libraries for data manipulation, geographical analysis, machine learning, and report generation. The core functionalities are organized into different sections accessible through a horizontal navigation menu.

**File Structure and Imports**

The code starts by importing necessary Python libraries:

* streamlit as st: The primary library for building the web application.
* streamlit\_option\_menu: For creating the horizontal navigation bar.
* visual\_graph: For generating graph visualizations (`generate\_visualization`).
* datetime as dt: For working with dates and times.
* base64: For encoding binary data (used for displaying PDF reports).
* pygeohash as gh: For encoding and decoding geohashes
* streamlit\_folium: For embedding Folium maps in Streamlit applications.
* folium: A library for creating interactive maps.
* numpy as np: For numerical computations.

The code then imports several modules from the python files in the ‘agile’ sub-directory:

* agile.filtering: Contains functions for filtering data based on location (`query\_location`), date (`query\_date`), ADID (`query\_adid`), and Overpass API node queries (`query\_node`).
* agile.mapping: Contains functions for displaying data on a map (`data\_map`).
* agile.locations: Contains functions for identifying locations of interest (`locations\_of\_interest`).
* agile.people: Contains functions for collocation analysis (`colocation`).
* agile.prediction: Contains functions for clustering locations (`double\_cluster`, `get\_top\_N\_clusters`).
* agile.utils.tag: Contains utility functions for finding nearby nodes (`find\_all\_nearby\_nodes`).
* agile.utils.geocode: Contains utility functions for reverse geocoding (`reverse\_geocode` - not explicitly used in this code).
* agile.utils.files: Contains utility functions for file operations (finding files, reading random lines, saving data, generating random names and aliases).
* agile.utils.dataframes: Contains utility functions for modifying and sorting DataFrame columns (`modify\_and\_sort\_columns`), and cleaning and verifying required columns (`clean\_and\_verify\_columns`).
* agile.profile: Defines a `Profile` class, likely used for analyzing individual ADID behavior.
* agile.report: Defines a `Report` class for generating reports.
* agile.centrality: Contains functions for computing location centrality (`compute\_top\_centrality`).
* agile.overview: Contains functions for generating data overviews (e.g., ADID value counts - `adid\_value\_counts`).
* agile.graphing: (AY 25 Addition) Contains functions for creating and manipulating graphs (`createGraph`, `connectNodes`, `connectCurrentNodes`, `expandNode`, `addADID`, `find\_cliques\_for\_adid`).

**Session State Management**

Streamlit's session state (`st.session\_state`) is used to store variables that need to persist across user interactions and reloads of the application. Think of these as global variables that exist throughout the webpage. These are variables that multiple pages/tabs will want to access and so you want to keep them consistent and the same across all of AGILE. They include:

* data: Stores the main DataFrame loaded from the uploaded file. Initialized to `None`.
* loi\_data: Stores the DataFrame of locations of interest generated by algorithms. Initialized to `None`.
* uploaded: A boolean flag indicating whether a data file has been successfully uploaded. Initialized to `False`.
* file\_source: Stores the name of the uploaded file or a flag if data was loaded from a demo. Used to prevent reprocessing the same file. Initialized to `False`. It also includes logic to delete previously saved data on startup.
* coloc\_ids: A DataFrame to store collocated ADIDs and their aliases. Initialized with columns 'Collocated ADIDs' and 'Alias'.
* generated\_reports: A DataFrame to store information about generated reports (ADID, Alias, Profile object). Initialized with columns 'ADID', 'Alias', and 'Profile'.
* alias\_ids: A dictionary to store the mapping between ADIDs and their generated aliases. Initialized as an empty dictionary.
* main\_adid: (AY 25 Addition) Stores the ADID that the user wants to further explore. Initialized as an empty string.
* radius, x\_time, y\_time, num\_nodes: (AY 25 Addition) Store parameters for graph generation.
* graph: (AY 25 Addition) Stores the generated graph object.
* lower\_bound, upper\_bound, has\_bounds: (AY 25 Addition) Store parameters for filtering graph edges.

**User Interface Layout**

The application's layout is structured using Streamlit's containers and columns:

* Title Container (`title\_c`): Holds the application title and logo. It uses columns to arrange the logo, title, and some spacing.
* Navigation Bar (`nav\_bar`): A horizontal menu created using `streamlit\_option\_menu` allowing users to navigate between different sections of the application ('Data', 'Graph', 'Filtering', 'Locations', 'Algorithms', 'Report').
* Sidebar (`sidebar`): The left sidebar is used for displaying controls and information specific to the selected section in the navigation bar.
* Blank Container (`blank`): Used for adding some vertical spacing.
* Data Options Container (`data\_opts`): Contains the 'Reset Data' button and the 'Keep Aliases' checkbox (though the checkbox is currently hardcoded to `True`).
* Preview Container (`preview\_c`): Displays a preview of the loaded data.
* Overview Container (`overview\_c`): Shows summary statistics and distributions of the loaded data (ADID counts, time distribution, geohash distribution).
* Results Container (`results\_c`): Used to display the output of analysis and filtering operations, including maps and DataFrames.

**Functionality Breakdown by Navigation Section**

**1. Data (`nav\_bar == 'Data'`)**

Data Upload: Allows users to upload a CSV file containing geolocation data with columns like 'datetime', 'latitude', 'longitude', and 'advertiser\_id'.

Demo Data: Includes a button to load demo data from `data/hainan.csv`.

Data Processing: (Upon upload or demo load)

- Cleans and verifies the presence of required columns.

- Converts the 'datetime' column to the correct format.

- Generates 'geohash' values if they are missing or have incorrect precision.

- Modifies and sorts the DataFrame columns.

- Generates random aliases for each unique ADID and stores them in `st.session\_state.alias\_ids`.

- Saves the processed DataFrame to a pickle file (`saved\_df.pkl`) for faster reloading.

Data Overview: Displays statistics about the loaded data, including ADID value counts, time distribution, and geohash distribution.

Alias Management:

- Check Alias: Allows users to input an ADID and view its assigned alias.

- Add Alias: Allows users to assign a custom alias to an ADID. This updates the `st.session\_state.alias\_ids` dictionary and the 'advertiser\_id\_alias' column in the DataFrame.

**2. Filtering (`nav\_bar == 'Filtering'`)**

Provides modules to filter the loaded data based on:

* Advertising ID: Allows users to enter an ADID and view only the data associated with that device.
* Location: Allows users to specify a latitude, longitude, and radius to filter data points within that circular area.
* Time: Allows users to define a start and end date and time to filter data within that time range.

The results of each filter are displayed on a map using `data\_map` and as a DataFrame in the `results\_c` container.

**3. Locations (`nav\_bar == 'Locations'`)**

Provides tools for analyzing specific locations:

* Node Query: Uses the Overpass API to find named locations (nodes) within a specified radius of a given latitude and longitude. Users can also specify a type of location to search for.
* Location Centrality Query: Identifies the most visited locations for ADIDs found within a specified radius of a given coordinate.
* Overpass Polyline Query: (Likely incomplete in the provided code) Intends to query points of interest along the path of a single ADID using the Overpass API.

**4. Algorithms (`nav\_bar == 'Algorithms'`)**

Offers various analytical algorithms:

* Top Clusters: Uses a clustering algorithm (`double\_cluster`, `get\_top\_N\_clusters`) to identify the top N locations of interest for a single ADID.
* Locations of Interest: Implements a traditional algorithm (`locations\_of\_interest`) to find significant locations based on the duration of stays and repeat visits.
* Colocation: Identifies other ADIDs that were present at the locations of interest of a target ADID within a specified time frame. Requires running a location of interest algorithm first.
* Movement Prediction: Uses a machine learning model (`Profile` class) to predict the location of an ADID at a specific time of day and day of the week. Requires training the model first.

**5. Report (`nav\_bar == 'Report'`)**

Allows users to generate a PDF report for a single ADID.

Report Generation:

* Checks if sufficient data exists for the selected ADID.
* Creates a `Profile` object for the ADID, which likely performs analysis to be included in the report.
* Generates the report using the `Report` class.
* Displays the generated PDF report within the browser using an `<iframe>` and base64 encoding.
* Stores information about generated reports in `st.session\_state.generated\_reports`.

Colocated Devices: Displays a list of colocated devices found for the ADID (if a report has been generated).

Generated Reports: Shows a list of previously generated reports.

**6. Graph (`nav\_bar == 'Graph'`)** (AY 25 Addition)

Provides functionality to create and explore a graph representing relationships between ADIDs based on colocation.

Graph Controls:

* Allows users to specify an initial ADID to query, a radius for proximity, minimum time difference for collocation (`x\_time`), minimum time gap for repeated collocation (`y\_time`), and the maximum number of nodes to display.
* Includes a button to 'Generate Graph'.
* Includes a button to 'Connect Current Nodes'.
* Includes a section to 'Expand Current ADIDs' by querying a displayed ADID.
* Includes a section to 'Explore New ADIDs' by adding a new ADID to the graph.
* Includes a section to 'Find groups for ADID' (clique finding).
* Allows filtering graph edges by weight.
* Includes a placeholder for filtering nodes by location.

Graph Generation: Uses the `createGraph` and `connectNodes` functions to build the graph.

Graph Expansion: Uses the `expandNode` and `addADID` functions to add more ADIDs and connections to the existing graph.

Clique Finding: Uses `find\_cliques\_for\_adid` to identify groups of ADIDs where every ADID is connected to every other ADID in the group.

Edge Filtering: Allows users to filter the displayed edges based on their weight.

Visualization: Uses the `generate\_visualization` function to display the graph using Plotly.

## 

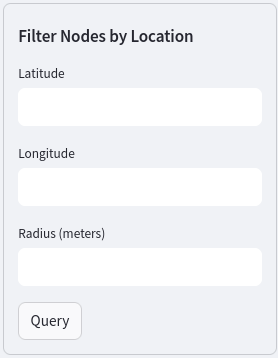
## **Where to Go From Here**

**Next Steps**

This section is meant to outline all of the features we wanted to implement next if we had had the time to do so. Our hope is that this will be a good place for you to start your contribution to AGILE. We will discuss small features that we think will make the tool more useful as well as the overarching next direction that the customer wants to see the project head in.

In terms of next features:

* Latitude/longitude query for the graph - Right now the graph is generated by focusing on the whole dataset of AdIDs and the user can also query graphs based on a particular Adid. However, the customer would like to also be able to produce a graph given specific geographic coordinates and a distance from that point. That graph would then display the relationships between people that have all been a certain distance from that point. You could also add a time criteria to narrow the search. In addition to making a new function in agile/graphing.py to produce such a graph, you would also need to add a new input form in the sidebar on the graph tab as shown below.



* Alias integration throughout the whole interface - On the first tab of the website you might have noticed that you can give AdIDs nicknames. This makes it a lot easier to refer to AdIDs as people, however, whenever you want to use any of the functions of the website, you have to use the AdID and not its alias. This is inconvenient because the AdIDs are a long jumble of letters and numbers. Therefore, it would be helpful if all the functions used the aliases as inputs as well as the full AdIDs. The aliases, if present, are also stored in the dataset, but this is a little challenging because the functions that are written within the agile directory all use the full directory. So they require constant switching between AdID and alias. Using a python dictionary could be very useful. If an Adid has an alias, add it to the dictionary. Then when using a function, if an AdID is inputted, then use that, but if an alias is provided, then use the dictionary to look up its AdID. The dictionary would have to be a session state variable in app.py so that it acts like a global variable throughout all the files in the agile directory.
* Filtering - Making the graph easier to filter would be useful. This could include:
  + Hide nodes on the graph that aren’t relevant
  + Hide specific edges that aren’t as relevant
* Chatbot - This one isn’t as crucial as the other features. But you could create a chatbot feature so that someone could type in what they want to know, and the chatbot would use the website’s features to produce an answer. This would be useful because people would require less training on the website to effectively use it. You could use open source AI like Llama to implement this.

In terms of larger scale improvements:

* Graph Neural Network (GNN) - We thought we could add a GNN to AGILE to help draw conclusions from the data. The truth was that we did not really know what a GNN was and how it could have helped us. We did not possess the expertise to implement the feature. However, now that we have built the graph, we believe that the GNN problem can now be tackled. You need to be aware of what a GNN will enable and how it can help with classifying edges.
* Scale AGILE - AGILE is meant to work on very large datasets, so it would be helpful if it were scaled up to work with hundreds of millions of data points. One place to start in order to achieve this would be to save graphs as objects. This way once a graph is made, it does not need to be made again, which will save computing power. Techniques like data indexing and efficient data structures could also be explored.
* Integrate tools together - right now AGILE has a lot of different tools that are located on separate tabs. And while they are all helpful and go together really well, because they are all in different places, navigating around from one to another becomes difficult and starts to detract from the tools themselves. To solve this issue, it would be very helpful if there was seamless integration/blending of the different tools and tabs for easier navigation and usability of the website. A way to do this could be to use the data and graph tabs as a home screen and then use buttons, like the ones that exist on the graph tab, to click on certain AdIDs to go more in depth and learn more. This would then use the features in the algorithm tab or filtering tab and then end with a more comprehensive report than exists on the current report tab. You could also have a “back” button to always be able to navigate a page backwards and a “return home” button to be able to restart.

**Challenges**

A few challenges worth noting is that the version of AGILE that you are working on is not a 1:1 copy of the version of AGILE that is used in the field. Our group did not do the DEVGRU internship the summer prior to first class year nor were we able to travel down to Dam Neck, so we never had a good grasp on how AGILE is actually used and what features would be helpful to them. If you do have those opportunities, then you will have a better idea than we did. In addition, due to the classified nature of the tool, we had difficulty predicting how our features may be used. Along with that, we lacked the computational power to test our work on very large datasets. And then finally, the previous group to use did not produce as in-depth documentation as we would have liked, so we spent a lot of our time just trying to understand how to use the tool and what all the code did. We also were not given a list of “next steps” by the previous group which also made starting work a little difficult, because we didn’t exactly know where to start. We hope this manual will solve that issue for you, if not, make getting up to speed a little easier.

**Lessons Learned**

Given those challenges, we wanted to share a couple of lessons we learned that will hopefully help future groups. Like we mentioned before, we started out with a lofty goal of implementing a GNN without having the required level of knowledge and experience on the topic. So, pursue the GNN, but ensure you have done the research. And then take time to understand everything before you tell the customer what you can deliver. It is okay to be realistic and tell the customer “no”. The customer will understand and being realistic about expectations is never a bad thing. That being said, still shoot for the stars and be ambitious.

Another big help for us was the usefulness of generative AI. We used a lot of ChatGPT to help us understand previously written code and to help us write new code. Our customer wants us to use generative AI, and using it speeds up the development process significantly. It gives you a great place to start, and then the code generated usually only requires some tweaking to get it working the way you want it to. It is also a great tool to look up how to do things you don’t know how to do. But note this, ChatGPT requires guidance and the user requires a strong understanding in computer science and AGILE principles. ChatGPT could not have done this project on its own.

Lastly, we learned how to work effectively as a team. We started by all working on the same part of the project at the same time, however, as time went on, we transitioned to splitting up the work and tackling the areas that fit our interests and expertise. We recommend that your group does the same.

## Appendix

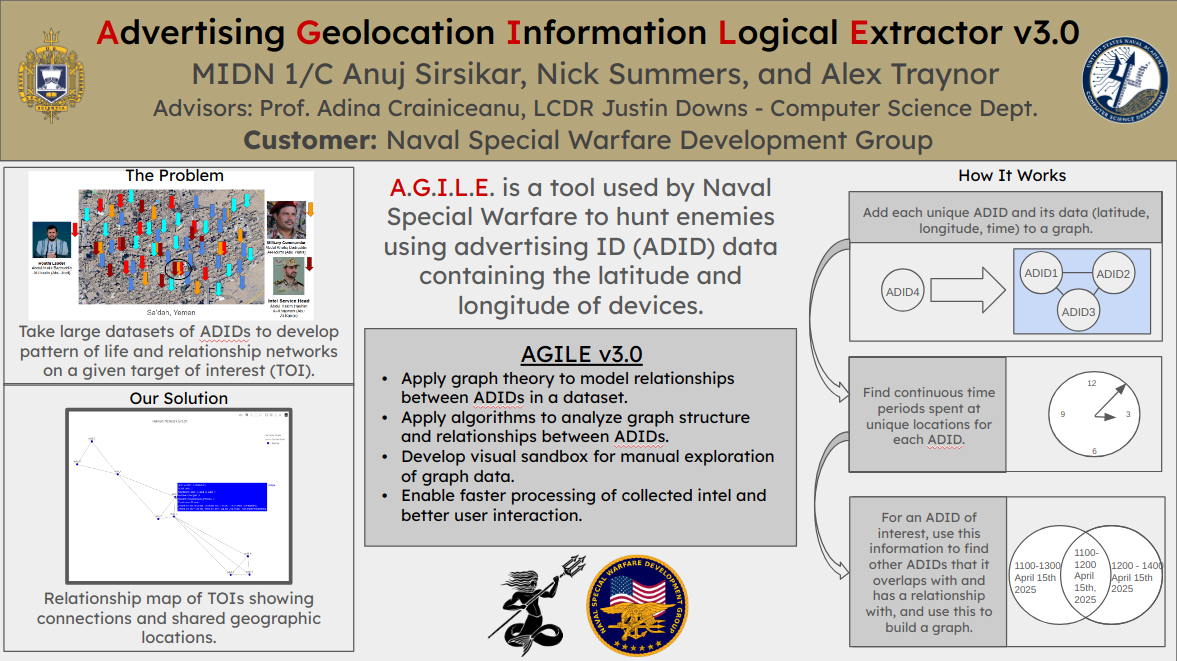
## 

## 

### Quick Reference Card

### 

### 



### System Requirements

### 

In order to download and run AGILE on your machine,you must have the following dependencies downloaded:

streamlit==1.20.0

proximitypyhash==0.2.1

pygeohash==1.2.0

pandas==1.5.2

matplotlib==3.6.2

Bokeh==3.0.2

streamlit-folium==0.7.0

geopy==2.3.0

scikit-learn==1.2.1

fpdf2==2.6.1

networkx==3.0

streamlit-option-menu==0.3.2

You must be running on a computer; currently we do not have support with mobile applications.

### Installation

### 

See **System Setup (Linux)**

### 

### 

### 

### Troubleshooting/Known Bugs

### 

### The only known bugs with the AGILE system that we discovered lie in the ‘Report’ tab. We never figured out how to generate a report as we always got an error when we tried to do so. We never really looked into it, because the customer said that the report works on their end and that they use it. You can attempt to fix it, but as long as it works on their end, it should be fine as is. You could revamp the report and make it more dynamic and not just a pdf. It could be interactive and go hand-in-hand with the graph and other features like the maps (and then be printable). That would be a way you could implement the integration of all of the tools together.

Another thing worth noting is that once you use the AGILE tools on one particular AdID, you need to hit the ‘Reset Data’ button at the top of each tab in order to use the tools on another AdID. If you ever get an error, start your troubleshooting by hitting that button. The 'Reset Data' button allows users to revert the loaded data to the state after the initial upload or demo load. It reloads the DataFrame from the `saved\_df.pkl` file.

### 

### Other than that, there are currently no other known bugs with the AGILE System and with any of the additions made by the AY25 group. While the project may not be complete as we are working on various further features to implement, the current model works as intended. The limitations are primarily due to large sample sizes of data which take a lot of time to process for the AGILE system. We have found that datasets with > 1,000,000 entries will cause the system to process for an inordinate amount of time. If the user chooses to implement less than this amount of data, then the system will process on a normal timeline.

### 

### Maintenance Procedures and Issues

### 

### We are in the maintenance and testing phase of our system requirements. Our procedures include continuing to maintain the system even after it has been presented. We will hand the reins over to members of the Class of ‘26 to continue great work on this exciting project.

### Developers’ Information

### 

Have questions for us? Feel free to contact members of the team below:

Nick Summers (Project Lead)

[m256144@usna.edu](mailto:m246252@usna.edu)

(817)-964-4522

Anuj Sirsikar (Team Member)

[m255838@usna.edu](mailto:m245370@usna.edu)

(978)-394-9426

Alex Traynor (Team Member)

[m256402@usna.edu](mailto:m246438@usna.edu)

(515)-777-5518

### Important Note From the Developers

### 

This guide, similar to the code, was written in large part by Gemini and ChatGPT (especially the parts where the code is explained in detail).