CSE 546 - DriveSafe

Group 35

Skanda Suresh | *Veda Vyas Miriyala* | *Venkatesh Polepally*

1217132644 | *12171150298* | *1217439197*

1. Introduction

Certain spots of roads have higher accident count due to various reasons. Few such reasons include, if it's a rainy day certain spots on a road can get slippery and treacherous to drive on, during night driving the visibility is very minute and there can be bumps and potholes on a road, tire blowouts because cause of bad road conditions and dust near construction sites leading to low visibility while driving [1]. Whatever the reason may be, these kinds of accidents can be prevented from happening if the driver is warned earlier about such risky road patches. To prevent such accidents, we have developed an application which uses GPS location to track the current location of the user on the road and keep him/her notified constantly and warns the user if a certain patch of the road is accident prone or not. Hence the user can be cautious and adjust his/her driving accordingly to avoid accidents. Our application infers the severity of warning by performing real time analytics based on the past accident history at that particular road patch the user is driving at.

2. Background

The technologies that currently exist are various Maps APIs and Geocoding APIs which support using a GPS location to detect a location of a person, translate it into address, show his/her the route to a certain destination and visualize them on a map. Few of such popular API choices available in the market are Google Maps API and OpenStreetMaps. Such technologies will help our application to track the location of the person from time to time and can be analyzed to infer if any person is nearing an accident prone area and notify them accordingly. On the other hand, any system is as good as the quality of the data it holds. So, we have researched various data sources such as Kaggle and Google Dataset search to collect reliable dataset for the accidents which can be analysed to derive accident patterns. This enables our system to be powered by data-driven decisions while warning the users. Last but not least, the most important feature of any application is how robust and reliable it is. To achieve robustness and reliability, we need the app to be easily deployable, easily scalable, highly available etc. All such things can be achieved in cloud computing environments. Therefore, we have decided to use Google Cloud Platform features. Google Cloud platform provides a wide range of features such as one click deployment, autoscaling , databases, queues etc.

Road safety is one of the most important concerns in the United States. In a majority of the road accidents, conditions such as the location, lighting, weather conditions such as rain etc. play a vital role. Analyzing past accident data to identify hotspots will make the road accidents more predictable and preventable. In efforts to prevent such accidents from happening, we propose a real time GPS based application which analyzes the past accident data to give out warning notifications to the user based on the route he/she is travelling. Based on the accident history of the roads, the app predicts the level of severity and warns the user about certain roads or stretches of roads which are in his/her route. Such

predictive warnings about accident prone roads might help users in being more careful and precautious thus decreasing the chance of occurrence of an accident.

Most of the current solutions relay a warning/alert to the user in the aftermath of an accident. Other existing solutions provide infographics describing accident hotspots. Our solution improves upon the previous approaches, leveraging predictive analytics based on past accident data serving real time notifications in a precautionary sense. The dynamic nature of road traffic patterns (depending on time of day, season of the year etc.) and huge volume of GPS data could impact the load on our servers thereby posing the need for a cloud computing solution capable of scaling in and out based on user demand. The current implementation on the market is if a vehicle is met with an accident the user logs in the details of the accident and the application remembers it. The next time the user travels nearby the accident area the application notifies the user to stay alert [2] . This existing solution has two major problems. Firstly, the app only notifies for an accident zone that has been logged in by the user and does not consider if the location is an accident prone area or not. As there might be many other vehicles that might have met with an accident at that particular location. Secondly, the app requires the user to log in the details into it which the user might tend to forget and the app would never notify back. In our solution both of these problems do not occur as all the accidents are evaluated to certify a location as accident prone or not and the app does not require the user to log in details.

3. Design and Implementation

In this section we present the various design choices that went into design and implementation of this application. Since the whole idea is based on GPS locations it would not be feasible to actually move on the road with the device in a vehicle. So, for the purpose of the demo and implementation we have simulated a car movement on the map.

Application Features and Workflow

As presented in Figure 1 the system has four major steps in the end to end workflow. Initially, the users should be able to enter a start and end location of their trip and choose the speed of the vehicle to simulate the vehicle movement. Based on the user chosen input, a simulated version of the vehicle represented by a watermark icon starts moving on the user chosen route visualized on top of a map. Finally, the corresponding GPS locations of the vehicle will be processed real-time by the system to notify the user with three levels of road safety warnings. The three levels of warnings are namely safe, moderate risk and high risk.

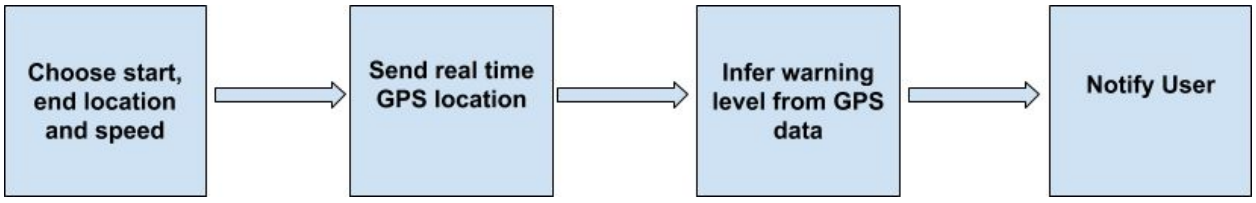

Figure 1: Application workflow

System Architecture

Our system follows a three tier architecture as shown in the Figure 2 The first tier is the client tier. On the client tier there is a web browser which offers a web UI through which the user can interact with the application. The second tier is the combination layer of application server and web server which

is hosted in a Google App Engine. The requests from the client's UI are received by the server over the websocket. The relevant information corresponding to the client's requests are extracted from the third layer called the database layer. This database is hosted on Google CloudSQL mySQL database. The extracted data is processed and upon completion the response is pushed back to the client over the websockets.

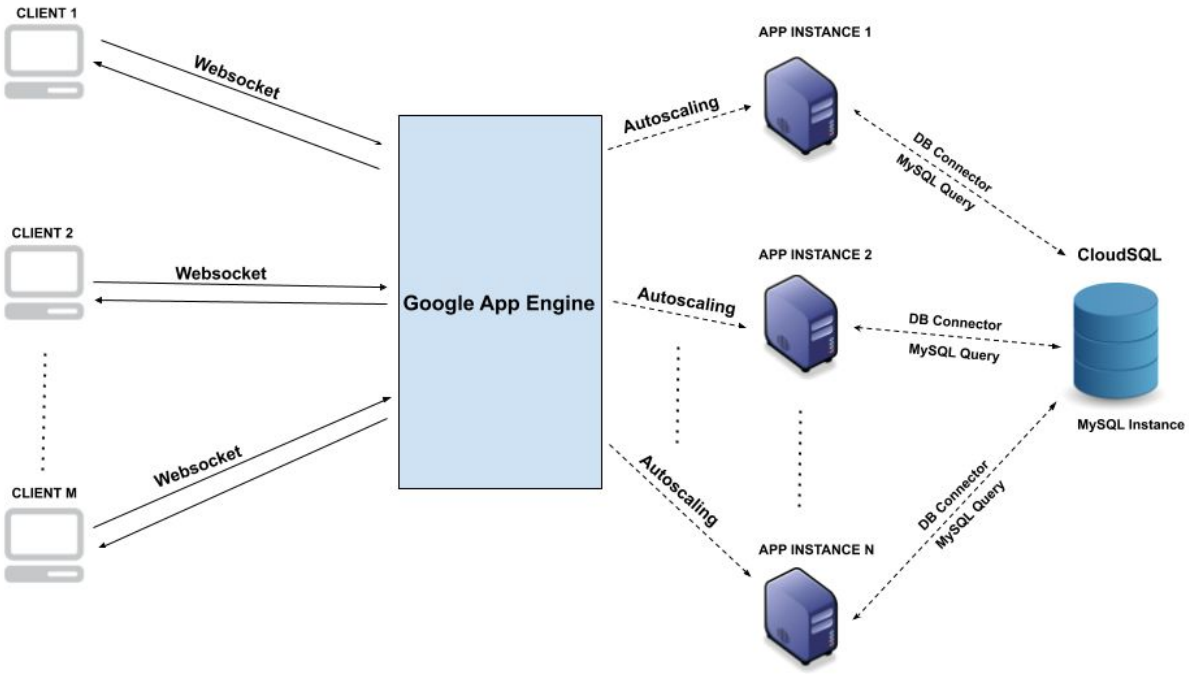

Figure 2: System Architecture

Client Side implementation

- Implemented using HTML,CSS and JavaScript. The home page is served by the Flask server as the home page of the app when the user hits the URL of the app.
- The user input of start location, end location and speed of the vehicle are captured using HTML text boxes.
- Then the *Google Maps Geocoding API* is used to geocode these start and end locations to latitude and longitude values.
- Then the *Google Maps Directions API* is used to retrieve the route between the start location and destination.
- Then the *Google Maps Javascript API* is used to visualize the extracted route and the vehicle marker on the map.
- A *Websocket* connection is established with the server which stays connected till the browser window is closed or till the destination is reached. Based on the speed chosen by the user, the

marker starts moving on the route and the corresponding GPS latitude and longitude are published to the server.
- When the server sends back a response with the road safety warning information, a colored icon is displayed where red indicates high risk, orange indicates moderate risk and green indicates safe.

Server Side Implementation

- On deployment of the application on the *Google App Engine*, a Python application is started and hosts a *Flask* web server.
- When the flask server starts, it establishes a connection with a *MySQL* database hosted on *Google CloudSQL* using an API called *SQLAlchemy*.
- For every incoming client connection a thread is automatically created by the Flask and it will start listening for messages. For every message received, the corresponding zip code of the latitude and longitude is reverse geocoded using the *Google Maps Geocoding API*.
- The zip code is used as part of the SQL query filter to extract accidents from the database. We also cache the filtered accidents data in-memory for each client until the zipcode doesn't change.
- Then using a search radius threshold the accidents are filtered and the accident count is calculated. The search radius threshold is determined based on the speed of the car and is designed to extract the area distance for the next 10 seconds so that the user will be warned earlier.
- The count value is classified into three levels of road safety warning using a thresholding mechanism. The inferred warning level is published back to the client over websocket.

Database Implementation

- U.S Accidents data set [6] from Kaggle is used for the purpose of the project.
- Dataset contains fields like the latitude, longitude, datetime, zip code, street, county, state etc.
- Since the state of California has the highest amount of data we have used that for our analysis.
- This data is loaded onto the *MySQL* instance on the *Google CloudSQL* service.
- For faster retrieval of the data we have created an index on the zipcode column.
- Also, we have done offline statistical analysis of the data to determine the threshold values for classifying into three warning levels.

WebSockets

We have evaluated various design aspects while implementing this application. One of the initial ideas was to use a traditional REST server architecture with message queues as the buffering mechanism. But we have later inferred that it is an overkill especially because even though we need the buffering for input, the stale warning notification outputs on the client does not matter as the client expects warnings for the live GPS locations only. Another major requirement was to maintain a continuous connection between clients to achieve a flexible two way communication. Finally, all these factors have made us consider websocket connections over the traditional REST calls. For the purpose of implementation, we have used flask_websockets library in Python and websockets library in JavaScript.

Google App Engine (GAE)

We have used Google App Engine(GAE) for the purpose of hosting our Python Flask application. Basically, it acts as a facade to both the aspects of web server and app server and thus playing a role of Platform as a Service(PaaS). I want to highlight three important aspects of our app from the perspective of the Google app engine.

First, since our implementation required websockets we are restricted to use the *Flex* environment because the *Standard* environment doesn't support websockets. Basically, Flex environment provides users with a wider range of abstractions compared to a standard environment and gives access to more lower level features as it is just as a wrapper on top of the Google Compute Engine.

The next important aspect is about the *Session Affinity*. Basically, when we enable autoscaling in a flex environment for websocket connections, some clients might do continuous polling to send messages and thus the load form the same client may get distributed over multiple instances. This may cause inconsistencies when some kind of state is retained. Therefore we have enabled session affinity as part of our application to ensure the messages from the same client are sent to the same app instance.

The final aspect is about how the whole UI and Server logic of the app is hosted on GAE and how the GAE features such as services and instances are used. We have created a single application/service and not multiple services as our app has only one core functionality of warning the user based on the GPS locations, so there was no need for microservice architecture. Depending on the autoscaling, the number of instances will be launched for the single deployed service. Also, the UI is not treated as a different application because our UI is very light weight. The main job of it is to render the map, display the marker and notify the user with a warning level. So we have included it as the static content as part of the Python Flask application itself. The Flask app is responsible to serve the index page when the user lands on the home URL. Once the static content is served by the flask serve end point, the dynamic content is rendered via JavaScript when required.

Autoscaling

Autoscaling features are different for flex environments and standard environments. Only auto_scaling or manual_sacling are supported for flex environments. We have used auto_scaling with min instances as 1 because we are using websocket connections one instance has to be live always. Also, flex mandates keeping at least one instance live all the while. Apart from this we did a scale test by opening a large number of client connections through browsers to understand the system utilization pattern. We have monitored these stats on the app engine dashboard to arrive at the value of 70% cpu target utilization parameter as part of autoscaling. Basically this parameter means that if the average CPU utilization across all the live instances is 70% then a new instance will be launched to handle the incoming requests. Also, I want to highlight that various parameters such as min_idle_instances, max_idle_instances are not supported in the flex environment.

Google CloudSQL

We have created a MySQL instance using the *Google CloudSQL* platform. CloudSQL is a platform which supports various types of RDBMS such as MySQL, SQLServer, PostGreSQL etc. The accidents dataset from the Kaggle is filtered for the state of California and cleaned and is uploaded into the accidents table in a MySQL instance in CloudSQL. To retrieve the data faster an index is created on the zipcode column. We have used *SQLAlchemy* API with mysql driver to connect to CloudSQL. There are two different environments we need the connection in. The first is when we are connecting from our local machine and the other is when we are connecting from Google App Engine. While connecting from local, we have used a proxy server to connect to the CloudSQL instance. To connect from the hosted Google App Engine application, we have enabled the linux sockets as we the application requires the usage of websocket connections in a flex environment. No special IAM roles are required as both the app and the database are part of the same project.

Google Maps Geocoding API

This API has been used as part of the project both in JavaScript and from Python. The primary functionality of this API is to gecode the the addresses to latitude and longitude or reverse geocode the latitude, longitude to addresses. In the UI, this was used to convert the user entered location to their corresponding latitude and longitude values. On the server side, it is used to reverse geocode the latitude, longitude values to extract pincode before querying the database for accidents data. We have used a Python library called GeoPy which provides a wrapper for most of the popular geocoding api's available in the market such as Google Maps, OpenStreetMaps etc. We have used the Google Maps implementation via the GeoPy library. The rate limit for this API can't exceed 50 requests per second.

Google Maps Directions API

This API has been used as part of the project on the UI in JavaScript. When the user chooses the start and end location, to simulate the vehicle movement on a route this API has been used. Since we are using the new account for Google Cloud we have received 300$ credit so we did not enforce any quota restrictions on the API.

Google Maps JavaScript API

This API has been used as part of the project on the UI in JavaScript. The primary reason to use this API is to display the map and render the motion of the vehicle on the map. Similar to other Maps API we did not enforce any quota as we had free credits in our Google Cloud account.

How does the proposed solution solve the problem?

Given the dynamic and variable nature of traffic patterns there is a need for a solution that can warn a driver of potential accident sites in a preemptive manner during a road trip. Our solution presents a data driven approach enabled by cloud technologies to solve this problem. Our solution leverages the statistical insights derived from large data stores of past accidents across a given region to provide the user with a cautionary signal depending on their current location. Our solution is enabled by the power of cloud thereby providing a system that is highly resilient, available and capable of scaling in and out depending on user demand. Furthermore, the flexible design of our system when combined with the comprehensive nature of the current dataset has the potential to be exploited for more complex data patterns thus allowing our solution to be extensible towards novel features.

Why is it better than the other solutions ?

Current approaches involve tackling the issue in a reactive manner in the aftermath of an accident. Drivers are notified only after the occurence of an event. Other systems involve users personally logging accident details, thereby sourcing accident data in crowd sourced manner. Our solution tries to build upon these approaches by a) notifying the user in a preemptive manner and b)Using comprehensive data obtained from a verified source to derive predictive insights. Our solution warns the user of a potential accident zone within ~10s. Such a system allows the driver to take appropriate control in time thereby mitigating the risk of an accident significantly. Furthermore our solution is cloud based, and is thereby highly scalable, resilient and available.

4. Testing and evaluation

The testing basically involved two phases, one was unit testing performed after development or configuration of a requirement and the other phase was integration testing which was performed once

the whole application was built. In the first phase the components involved were the User Interface code, the JavaScript code to connect to the backend, the backend server code with database connection, google app engine with auto scaling testing.

- The user interface code testing involved the testing of the map feature that was embedded in our UI to check whether all the functionalities in it are working as expected and if there is no problem when opened in multiple tabs as we're using an API key. Testing the input fields and notification bar if it updates properly when an event is triggered. Testing the marker updates on the map when a user is on the move.

  The evaluation results for the tests seemed to fetch a few bugs which were fixed and retested until all the tests passed.

- The testing in the JavaScript side involved if the speed input was working as expected to move the marker at the expected speed, the marker location movement testing, websockets testing to see whether the message are sent with their id's as expected and received as expected when sent back from the server and also updating of the notification system appropriately when a message is received.

  The evaluation results for the tests seemed to fetch a few bugs which were fixed and retested until all the tests passed.

- In the backend code, the database connection and fetching of the data was tested with time taken to fetch the data as we were motivated to build a real time response system, testing the logic to find out the zip-code of a location, testing the logic to categorize a location into green, orange or red zones according to the criticality of accidents in the zone and also the websocket connections to the frontend with send and receive data functionality. All of these functionalities were tested for time complexities as well to improve it to as better as possible to deliver accurate real time updates.

  The evaluation results for the tests seemed to fetch a few bugs which were fixed and retested until all the tests passed.

- To test the google app engine deployment, we tested it initially by deploying our prototype application and testing to see if the request and response data are being transferred at pace. This testing helped us set a few configurations which we initially did not for the better working of the app. We integrated the google relational database and tested the connections. After enabling the auto scaling we tested it by launching multiple clients to send requests to the backend and check in the google console if the instances are being scaled according to the requests in the app engine.

In the second phase of testing we performed integration testing to test the whole application. All the components were integrated together with integration testing done at every phase of integrating 2 components and any issues found were resolved then and there. After the whole application was built and deployed end-to-end testing was done checking the logs and time taken by each component was tested to see if it lies in the required range. Manual testing of the application in a single browser window was performed by changing the input values on the UI screen and testing if the generated output was as expected by us and also tested the same by launching the application on multiple browser windows and testing all the features of the UI concurrently. The results yielded an issue where while launching multiple clients we faced breakage of the code due to a google database configuration which wasn't set, the change was made and while we retested the application seems to work smoothly.

5. Code

APP LINK - http://drivesafeapp.wl.r.appspot.com (Please use only HTTP and not HTTPS)

Run and deploy locally

1. Unzip the folder and cd into the folder.
2. Install Python 3
3. 'pip install -r requirements.txt' to install all the libraries
4. Add the relevant database such as db username, db password in the CloudSQLDAO file.
5. To connect with CloudSQL database we need a proxy server to be stetup.
    a. Follow this guide - https://cloud.google.com/sql/docs/mysql/quickstart-proxy-test
    b. curl -o cloud_sql_proxy https://dl.google.com/cloudsql/cloud_sql_proxy.darwin.amd64
    c. chmod +x cloud_sql_proxy
    d. gcloud application-default login
    e. ./cloud_sql_proxy -instances=drivesafeapp:us-west2:drivesafe-app-db=tcp:3306
    f. NOTE - setup your own database as the proxy server authenticates through gmail login in step C.
6. Launch server using: gunicorn -b 127.0.0.1:8080 -k flask_sockets.worker main:app
7. Launch the browser and enter localhost.com:8080 to access the app.

Run and deploy on GAE

1. Setup GAE project on GCloud console
2. Use the existing app.yaml but change the env values to your own values for DB authentication
3. Install gcloud sdk on the machine
4. Then do 'gcloud init' to setup, 'gcloud app deploy' to deploy the app
5. Then launch 'gcloud app browse' to access the applink in the browser

Code Explanation

- **index.html** : The index.html file contains the user interface code which basically contains elements used for forming the input element, start element and code for the notification bar. The Html file also has an embedded map which is linked to google maps api. The style for all the html is in style.css.
- **Script.js** : The script.js (developed upon [7]) has all the JavaScript code related to update and retrieve data from the client's UI. It has code to calculate the speed the user enters and move the marker at that rate, also the map related logic to find the route between two entered locations, the start and the destination location and movement of the marker from the start to the destination at the mentioned speed. It contains code for forming a websocket connection with the backend and communicating through it, by sending the location details at intervals to the backend and also receiving back the json from the backend and displaying the notification based on the received data.
- **CloudSQLDAO.py:** Defines a CloudSQLDAO class with *constructor* and *get_accidents_by_zipcode* functions
    - *Constructor* : Creates an sqlalchemy engine instance to connect to the CloudSQL database with appropriate authentication. Authentication is determined based on whether the call is made from Google App Engine, in which case authentication is not required, or from a 3rd party instance in which case appropriate username and password must be entered.

    - *get_accidents_by_zipcode* : Retrieve a filtered list of accident sites based off a zip code key. This filtered list is used to optimize the search space in real time analysis.
- ***app.yaml :*** This file provides the runtime configuration of the Google App Engine. Some of the important attributes present in this file are:
    - *runtime* : Instance of App Engine (python in our case).
    - *runtime_config* : version of instance (3).
    - *env* : Type of GAE environment being used (flex in our case).
    - *entrypoint* : Specify command which is run when GAE launches our application. We have used the gunicorn launch command which supports flask web sockets.
    - *env_variables* : Environment variables set in GAE environment. Used for database authentication.
    - *beta_settings:cloud_sql_instances* : Used to refer to database instance connection name.
    - *automatic_scaling* : Here we provide auto scaling parameters, such as min_num_instances(minimum number of instances), cpu_utilization:target_utilization (threshold utilization for scaling up).
    - *networks:session_affinity* : When set to *True***,** used to drive messages from single client to a single server.
- **utils.py :** Helper functions that are used by main.py to compute warning signals given a GPS location. Functions are as follows:
    - *infer_warning_score* : Given a set of gps coordinates it computes how many of them lie within a particular threshold distance of the current location using the *get_points_within_radius* function. It returns the number of such accident sites as a warning score.
        - *get_points_within_radius* : For a given row in a dataframe of GPS coordinates, it flags whether the location lies within a particular threshold of a query location (1/0).
    - *reverse_geocode* : Function to convert a given GPS location into a human readable formatted address using google geocoding api. Appropriate exception handling logic is incorporated to ensure delays when Rate Limit of api is exceeded, this allows the system to function smoothly.
- **main.py :** Core server side functionality resides in this program. It receives real time GPS data, parses the formatted address using reverse geocoding and invokes the controller functionality wherein it queries a filtered set of accident locations from the CloudSQL database using the zipcode as a key. The functions present in it are as follows:
    - *socket_connection* : This function receives an incoming message, i.e the real time GPS coordinates using socket connection. It then uses *utils.reverse_geocode* to convert GPS into formatted human readable address objects. Next it extracts the zipcode from the address object and uses *CloudSQLDAO.get_accidents_by_zipcode* to query a filtered list of accident sites using zipcode as the key into a dataframe. This dataframe is further processed using *utils.infer_warning_score* to identify the number of nearby accident sites using a statistically predetermined threshold. Finally this data is aggregated and sent back as a warning signal to the client along the same socket connection.

6. Conclusion

In this project we've accomplished to solve a real world problem by applying the concepts we have learnt as part of this course to design, develop and deploy a cloud based web application from scratch. In the process of the development of this application, we gained exposure to a vast breadth of concepts in technologies corresponding to User Interfaces, Cloud computing, Networking, Databases etc. Few such takeaways from this project are using HTML, JS and Google Maps JS API to visualize maps, and vehicle movement on the route, using websockets and socket programming to maintain a constant connection with between the client and server, using the PaaS based cloud computing technology such as Google App Engine to deploy and autoscale our application, using and integrating other Google cloud based services such as CloudSQL, Geocoding API etc.

Our current solution is tested only on the state of California as we could only get the quality accident data for that particular state. Whereas our design and implementation choices such as zipcode based indexing, autoscaling makes our solution capable of scaling to any state, country in the world. However, there are better GeoSpatial indexing structures such as Grid based indexing, Hotspot based indexing etc. which can be leveraged along with the zip code based indexing to create multi level indexing structures. This kind of design would not only help in faster data retrieval but also helps in retrieving only a fewer rows thus making the radius based search faster. Another area of improvement can be achieved by factoring various parameters of the accidents such as the side of the road, the road name, the time of the accident along with accident count while performing the warning level analysis. Most importantly we believe that the true value of the solution does not arise from the prototypes, proof of concepts alone rather it arises from productionizing the solution for the real world usage. Therefore, we believed that this app could have a ground-breaking impact by saving millions of human lives by preventing road accidents if deployed for real world usage by utilizing the real-time dynamic data of accidents which can be fed into our system by exposing an API to United States authorities such as DMV etc.

7. Individual contributions (optional)

Skanda Suresh (1217132644)

My contributions are primarily centred around the design of the search algorithm for GPS coordinates to aggregate nearby accident sites in order to generate a potential accident warning signal. I also contributed to integrating the geocoding api, offline statistical analysis and testing/evaluation procedures.

GPS search space filtration

GPS coordinates are relayed in a realtime fashion from the client to the App engine instance. Given a GPS coordinate the server must find all recorded nearby accident sites within a particular threshold distance. The dataset we are using is very large in size, it has approximately 250,000 recorded accidents in california. Running a distance computation function on a dataset of this magnitude would not be feasible in a real time application, thus there is a need to optimize the search space. I helped develop an indexing mechanism based on the zipcode to ameliorate this. Each accident site's GPS location can be reverse geocoded into a human readable address. The zipcode can be extracted from this address, and I helped develop a parser to do this. Once the zipcode is extracted, we can group accident sites by zipcode thereby creating an index by zipcode. This way the zipcode of the real time coordinate is used as a key,which we use to retrieve a set of filtered records from the CloudSQL database. This filtered set is the set on which the search computation is carried out. This significantly helped reduce latency.

Geocoding

A major challenge in our project involved parsing GPS coordinates to human readable addresses and vice versa. We make use of google's geocoding and reverse geocoding api to do this. These api's came with limitations to usage in the form of RateLimit and ServiceLimit constraints. I helped in exploring various apis, testing them to their limits and finally decided upon the suitable service to be used in our application. I also helped write the exception handling logic for these services in the event of a RateLimit exceeded / ServiceError. This logic was written to ensure a seamless performance while abstracting away the error handling latency to the user. After experimentation we decided to use the google maps api via the geopy wrapper class.

Statistical Analysis

Our dataset consists of a large number of accident reports with different levels of severity provided. The optimal search algorithm makes use of a threshold radius to count the nearby accident sites. This threshold is determined by offline statistical analysis. The offline statistical analysis consisted of clustering the GPS locations and identifying hotspots. These hotspots were further analysed and we computed the average number of accidents by cluster. These computed metrics were analyzed and visualized to identify a suitable threshold for the state of California.

Testing and Evaluation

I contributed to the unit testing procedure of the app engine. I ran test cases to identify when the socket connections broke. I especially helped in identifying RateLimit errors via geocoding apis by analyzing the logging information. I also helped monitor app performance after the entire solution was integrated, and analyze the stability of the SQLAlchemy pool allocation.

Veda Vyas Miriyala (1217150298)

Design

From the design aspect the challenges were to find out what components were required in the system, what resources to be used. Initial steps involved, evaluating various Google cloud services and creating free tier accounts for 300$ credits. I was involved in designing the system architecture. Where we compared and discussed pros and cons of both Socket connections and REST server based architecture with message queues. After we decided to go with web socket connections, I was involved in verifying the compatibility of the websockets in the Google App Engine environment. Also, I was involved in the setup of various Maps APIs and Also, I was involved in evaluating database design choices. We have evaluated various design choices such as RDBMS or NoSQL and we then finally went with the SQL based design as we have structured data. Therefore, I was involved in exploring various RDBMS options from Google Cloud Services.

Implementation

In the implementation phase, I was involved in the implementation of Google Cloud App engine aspects. I have worked on creating an app engine project, especially deploying a Python application. Researching various types of environments such as Flex and Standard Google App offers. Worked on exploring Python libraries for websockets. I was also involved in creating a web server for which I have used Flask as the framework and gunicorn as the hosting server. I have also implemented the websocket connections using flask_websocktes library in Python. Also researched on various auto scaling parameters for the flex environment and tested them in various scenarios. This was the most challenging part as the parameters are quite different from the popularly used Standard environment. Also, I was involved in the understanding and cleaning the accidents data set. I worked on setting up the database and pushing the data into the Cloud SQL database. Worked on various other aspects of databases such as creating database schema, tables and index structures. Used sqlalchemy mysql as the connector object to connect to the database and also implemented a connection pool to maintain parallel connections with the database. I have also implemented the data access object(DAO) code in Python to perform various SQL queries on the database. Also, I have worked on setting up connections to CloudSQL from both local machine and Google App Engine. For the local connection part I have worked on setting up a proxy server which can perform the authentication to connect to the CloudSQL. Also, in the GAE setup I have worked on setting up the unix sockets connections via the app.yaml.

Testing

In the testing phase I was involved in both unit testing and integration testing. Most importantly my primary focus was on unit testing the features that I have implemented. Few such things are, I have worked on testing the reliability of the web socket connections and experimenting with a latency involved. Also, I've worked on testing the database pool connections dropout and cleanup. Also, I have worked on testing various configurations for autoscaling and testing the performance for each of the configurations. I have also worked on testing the application end to end, especially looking out for various start and end locations to validate whether the app is working fine or not and also checking how the app behaves in the known accident prone zone.

Venkatesh Polepally (*ASU ID - 1217439197*)

The project aims at using the platform as a service cloud computing technology to deploy a self built real-world application and automatically scale it based on demand. The resources user were the Google App engine, Google maps, Google cloud SQL. It is called Drive Safe application, which notifies users regarding accident prone areas while they are traveling. The application exposes a user-interface for clients to access. So the tasks involved were designing the architecture, implementation and testing. Below are my contributions to each of these phases.

1.Design

In the design phase the first step we took was to decide which resources we will be using in our application and how they're going to be architected. The discussion regarding the communication between each component in the structure. Whether to use Http requesting mechanism, queue based requesting mechanism or WebSockets. I worked on finalizing the design of the client facing application and the rest service in the cloud. I prototyped multiple ways of communication such as using Google task queues, WebSockets, Http connection etc. Calculated the latencies and selected the best among them to be used in our application. Setup a google cloud account to use these features. Designed the Client facing application. Subscribed to google's map API, to use features like directions API, current location API etc. Referred multiple IEEE papers to gain knowledge on the technologies used and used the ideas gained to architect and develop our project.

2.Implementation

In the implementation phase, I implemented the initial prototype where we used a node server and node client to make them communicate with each other in different forms of communications such as WebSockets and queue based communications. I implemented the Complete User interface screens with the UI elements to send and receive data. Integrated the Google map API to our application to use its API's. Wrote the logic for demo simulation code wherein a marker is moved from a source location to the destination location at the mentioned speed of the user. Read the documentation for Google cloud to help in setting up the App engine. Integrated the backend Flask Code with the frontend javascript interface. Learned about how to use Cloud SQL and contributed to the implementation. Documented the Report for this project.

3.Testing

The testing basically involved two phases, one was unit testing performed after development or configuration of a requirement and the other phase was integration testing which was performed once the whole application was built. In the first phase I worked on components where I did the implementation and also tested everyone else's components to find bugs if any. The components involved the client facing user-interface code, the socket connection code in the UI and backend, The SQL connection and query code, the backend business logic code. Once the components were developed at each stage of integration we tested the components again to see if there is no breakage. And finally after the complete integration of the application we tested the application for multiple variations in the UI input fields, launched multiple clients in multiple computer systems to scale up and down the app engine code and check whether the auto scaling feature works and yet being able to serve the user interface code with real time location data.

8. References

[1] “Top causes of car accidents and how you can prevent them” Accessed on:April. 30, 2020. [Online]. Available : https://www.huffpost.com/entry/top-15-causes-of-car-accidents_b_11722196

[2] “Accident prone location notification system and method” Accessed on:May. 1, 2020. [Online]. Available : https://patents.google.com/patent/US8289187B1/en

[3] “Accident prone area detection using driver stress database” Accessed on:April. 36, 2020. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/8325994

[4] “Prediction of the cause of accident and accident prone location on roads using data mining techniques ” Accessed on:April. 30, 2020. [Online]. Available : https://ieeexplore.ieee.org/document/8204001

[5] “Analysis of historical accident data to determine accident prone locations and cause of accidents ” Accessed on:April. 27, 2020. [Online]. Available : https://ieeexplore.ieee.org/document/8699325

[6] “US-accidents-dataset” Accessed on:April. 28, 2020. [Online]. Available : https://www.kaggle.com/sobhanmoosavi/us-accidents

[7] “Vehicle movement simulation” Accessed on:May. 6, 2020. [Online]. Available : https://github.com/AvinashKrSharma/vehicle-movement-animation-google-maps