University of Arkansas – CSCE Department
Capstone II – Final Report – Spring 2023

Ponga Software Usage Tracking

Matthew Clemence, Dylan Vaughn, Ryan Drake, Julio Bonilla, Logan Reed

Abstract

In the world of software, the necessity of user data tracking is evident. Ponga wishes to nudge its users towards certain value propositions in order to increase their lifetime value. These value propositions comprise the number of uploaded photos, the number of shares, and the number of comments one user perpetrates. With a healthy mix of these three value propositions, a user's lifetime value would be considered satisfactory. The amount of time a user has existed also comes into play, but the numbers of their value propositions wouldn't reach a significant high without an extensive relationship with Ponga.

Our objective is to design a series of AWS Lambda functions and a database structure to handle the tracking of users. The data gathered on each user will be drawn on to assess their risk level as a customer and Ponga will act accordingly. To determine a user's risk level, we will perform calculations with a user's value proposition numbers. If the user's numbers are unsatisfactory in one area, then Ponga can push the user in a different direction. The significance of this problem lies in our unique solution to it. Our software is named Insight and will be unique in tracking user data by comparing value propositions over time via calculations involving n-dimensional geometry.

1.0 Problem

Ponga is a web application for photo identification, preserving family stories, and keeping track of people, places, and things. Ponga took off when they implemented a membership model on top of an already richly featured platform. Features include the ability to: highlight people and/or details in a photo with a zoom and click, attach a link to a photo, and play music alongside a photo. Ponga is for private interactions among families and not advertisers. Ponga is delving into the process of advertising themselves and that's where Insight comes in.

Every product that exists today embraces some sort of advertisement to bring in more customers. Ponga wants to not only bring in more customers but reinforce the ones they have. This can't be done without gathering appropriate data on their customers which portrays their use of the application over time. Ponga's problem can be demonstrated through a simple product such as Reese's. Reese's have two value propositions also interpreted as groups from which their customers come from. There is the group who enjoys chocolate, and the other group who enjoys peanut butter. For Reese's to be successful they must be able to increase the number of customers who belong to both groups. Reese's problem is the same as Ponga’s in that they need to know which customers lean towards one or more value propositions over others.

This is a problem every product on the market runs into, be it an application or candy. Not defining value propositions and grouping the customers by them can lead to stagnation in a product. Risky customers are those who do not fall into all groups of a product and as such are more likely to leave said product for another. In contrast, a healthy customer is one who falls into all groups and their respective scores in those groups are determined by the product. Without a deeper examination of their customers, Ponga can’t influence risky customers into becoming healthy ones.

2.0 Objective

The objective of Insight is to design a solution that records customer activity, quantizes their activities, and generates meaningful data that can be used to analyze a customer's level of risk within an arbitrary window of time. Insight aims to track any change in customer behavior over time and advise Ponga about any potential customer risk, trends, or anomalies. Our design will involve a series of AWS Lambda functions coupled with DynamoDB as a database. One will retrieve data from a database table, and another will send data to a separate database table. Lastly, one will calculate user attribute affinity based on the data sent and fill the table to be retrieved from. Our team aims to resolve the problem outlined above by instantiating APIs that are triggered by a user making any action on their account and organize that data so that Ponga can better manage that user.

The initial goal is not to weight value propositions, but to observe and collect user data; thereby allowing a thorough analysis of user patterns, preferences, and engagement levels. Using Amazon Kinesis Data streams, all data moving through our model can be timestamped and stored in each user record. The AWS platform will allow us to integrate services necessary to maintain a service level agreement prescribed by Ponga. The team will research the AWS platform and familiarize themselves with its functionalities. The team will also synthesize a solution to compare and quantify data points within an nth dimensional vector space. The team will remain in communication with the POC on a weekly basis to ensure we are operating within desired metrics.

3.0 Background

3.1 Key Concepts

The meat and bones of Insight will be created using AWS (Amazon Web Services). AWS is a cloud provider that provides servers and services that can be used on demand and scales easily. AWS powers some of the biggest websites in the world, one being Netflix.

[1] AWS Lambda is one service we will be using. AWS Lambda introduces the idea of a serverless experience by employing virtual functions. Rather than being limited by RAM and CPU, these functions are limited by time. These functions are run on-demand, meaning there is

no charge if the functions are not being used. When these functions are being used, we are billed by the 100ms. The most significant feature of AWS Lambda is automatic scaling. This means that as many instances of a lambda function that we need can be run all at the same time. This service cuts out the need to manage the infrastructure of an application and allows us to focus solely on what the application does.

[2] DynamoDB is another amazon web service we will utilize. DynamoDB is a managed NoSQL database optimized for performance at scale. NoSQL stands for Not Only SQL. DynamoDB acts as a key/value lookup store, meaning values that are stored in the database are found through an API given a key. This is slightly different from the traditional SQL database with only rows, columns, and tables. A managed database is one with the storage, data, and computing services maintained by a third-party provider. DynamoDB can scale out horizontally by adding more nodes onto the cluster and separating data out onto those nodes. This means that even if the size of the data grows, the performance will not see a major change. DynamoDB is a very safe service to use due to its high availability and durability. For availability, DynamoDB has a 99.99% guaranteed up time which translates to around 26 seconds of monthly down time. For durability, DynamoDB stores multiple copies of data across multiple nodes to avoid loss of data. One caveat of DynamoDB is that one should know how they want to access their data before using it, otherwise you may not be able to access data in the way you want when all is said and done.

[3] Amazon API Gateway is another service needed for Insight. With API Gateway, we can create, publish, maintain, monitor, and secure REST, HTTP, and WebSocket APIs at any scale. We will use Amazons API Gateway because it will easily integrate with other web services such as Lambda and DynamoDB. Our application will need to receive API calls from the Ponga application, and this is done through Amazon API Gateway.

[4] Amazon Kinesis allows for incoming data to be processed in real time without interrupting the database in any way that would prohibit other actions from being completed simultaneously. Using this, Insight would be able to analyze incoming data and trigger Lambda to begin working on manipulating data and sending it to a separate table. When the data arrives in the second table, Kinesis could trigger another action.

[5] Amazon EventBridge is a serverless event bus service that you can use to connect your applications with data from a variety of sources. EventBridge pipes deliver a stream of real-time data from your applications, software as a service (SaaS) application, and AWS services to targets such as AWS Lambda functions, HTTP invocation endpoints using API destinations, or event buses in other AWS accounts. We will use EventBridge pipes to send a DynamoDB data stream to multiple API endpoints triggering their own AWS Lambda functions.

[6] Amazon Simple Queue Service solves the problem of requests from the client to the application being lost due to the workload. Data sent and received between the client and API Gateway will be processed through SQS. This allows an application to take as many requests or data sent from the client as needed and process them when the API Gateway is available. This ensures that none of the client data is lost, or the API Gateway doesn’t incur an error from an overload. The SQS allows for a manageable workload at any scale.

3.2 Related Work

Insight falls under the field of user data tracking applications. The most common ways to track user activity today include tools like Google Analytics and Search Console, click tracking, scroll tracking, and viewing session recordings of users. These methods all track user activity on a website or app in a rudimentary way that requires excess effort from the client.

[7] Google Analytics is a platform that collects data from websites and apps, then provides reports to the client. Setting up Google Analytics for yourself requires making an account and adding JavaScript measurement code to each page on your site. This tracks how many users visited a particular page or bought a certain item on a store page. It collects user browser information such as: language, type, device, and operating system. Once this data is collected by the JavaScript code, it is packaged and sent to Google Analytics to be processed into reports. These reports contain the aggregated data organized by user device and browser. The client can also filter the data sent to Google Analytics if they wish to protect private company data. The processed data is stored in a database where it can't be modified. This could be used to gather the data for our problem, but the client still needs to analyze it themselves.

[8] Click tracking is exactly what it sounds like. All user's clicks on a website are tracked and compiled into a report for the client. Clicks tracked include those on links and buttons. This is useful for website design and marketing. The reports contain events with click being its own event. The number of clicks per user is displayed, and each button/link has its own value. [9] Scroll tracking tracks how far a user scrolled down a web page. A trigger called the scroll depth trigger is used to track this. There are vertical and horizontal scroll depths. The client can enter percentages representing the page height or width to set trigger points on the page. The variables saved for future reference are scroll depth threshold (depth of trigger in percent), scroll depth units (pixels or percent), and scroll direction (vertical or horizontal). While our problem deals with marketing, we are tracking the use of core elements of an application which includes more than buttons and links.

[10] Session recordings are renderings of real actions taken by users as they browse a website. Recordings capture mouse movement, clicks, taps, and scrolling data for desktop and mobile devices. This kind of tracking can help improve the user experience of websites and applications. By watching session replays, the client can find patterns that explain the prevalence of risky users. Recordings also help catch any bugs causing users to get stuck, confused, and frustrated. The problem here is that the client must watch these recordings and spend time making sense of their customers' actions. For a large-scale application, watching all the session recordings and putting together a trend would be more work than it's worth.

Overall, the problem with Google Analytics and session tracking is the amount of interaction required by the client and the nature of the processed data. Google Analytics just sorts the data for easier readability, but any trends need to be inferred by the client after receiving. Our application focuses only on the value propositions of an application/website and the user's trend with them. The client will provide as many value propositions as they want and connect the quantifiable actions taken to our API endpoints. Our processed data will determine a trend towards one or more value propositions. Once the client gets a read on their average customer, they can then add weights to the value propositions with triggers to notify the client when users enter a risky state. The session recordings would work for Ponga, but it is a rough solution to a long-term problem. The difference between our data tracking and existing methods is the enhanced focus on value propositions and the use of N dimensional geometry for calculations on behalf of the client.

4.0 Approach/Design

4.1 Use Cases

Use Case: Submit Updated Metric
Author: Jerrel Fielder
Primary Actor: CLIENT application via *POST* method to API URL
Goal in Context: To provide an update to one or more user engagement metrics for tracking.
Preconditions: API should be available, and the call should be formatted correctly.
Trigger: The user takes a quantifiable action in the CLIENT application and the CLIENT invokes the API in order to log the action.

Scenario:

1. User takes quantifiable action in CLIENT application, such as uploads photographs
2. CLIENT counts the number of subject items in the user action
3. CLIENT invokes API URL providing:
    a. User: <email>
    b. JSON payload with an array of at least one key-value pairs:
        i. Attribute ID: count
4. If the CLIENT passes a key belonging to a customer that doesn't exist in the system, the system must create a record for that primary key and the data from CLIENT call becomes the T_0 data point.
5. SYSTEM uses current system time as the timestamp to be persisted with the event in the database

Exceptions:

1. Successful call returns HTTP status code 200
2. Incorrectly formatted API call from CLIENT returns HTTP status code 403
3. Unauthorized access attempt returns HTTP status code 404
4. Server error / down returns HTTP status code 500

Priority:	High
Channel to actor:	CLIENT application
Frequency of use:	Up to many thousands of times per day.
Secondary actors:	
Channels to secondary actors:	

Open Issues:

Use Case: Retrieve Historical Data for a User
Author: Jerrel Fielder
Primary Actor: CLIENT application via *GET* method to API URL
Goal in Context: To retrieve the historical attribution of a user.
Preconditions: API should be available, and the call should be formatted correctly.
Trigger: CLIENT application requires the historical attribute trajectory for internal processing and/or visualization. Results should always be presented in reverse chronological order (newest first, oldest last) and limited to the number of records specified in the OPTIONAL parameter "limit: n". If no "limit" parameter is included in the call, return all records in batches of 50 with a pagination ID in the return set.[1]

Scenario:

1. User is logged in and active in the CLIENT application
2. CLIENT needs raw historical data (visualization, etc.)
3. CLIENT invokes API URL providing:
    a. User: <email>
    b. OPTIONAL JSON payload specifying maximum array elements to return:
        i. limit: n
4. Response Payload:
    a. User: <email>
    b. Records: <n records>

```
[
  {
    i.  Timestamp: <created date>
    ii. Data: <Magnitude, All Angles>
  },
  …
  N
]
```

Exceptions:

1. Successful call returns status code 200
2. Incorrectly formatted API call from CLIENT returns status code 403
3. Unauthorized access attempt returns status code 404
4. Server error / down returns status code 500

Priority:	High
Channel to actor:	CLIENT application
Frequency of use:	Up to many times a day.

[1] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.Pagination.html

Use Case: Calculate Attribute Affinity and Persist
Author: Jerrel Fielder
Primary Actor: System
Goal in Context: To calculate the current attribution affinity for a given user.
Preconditions: None
Trigger: Upon the update of a user (via the “update User” API), calculate the current attribute affinity and update the user record.

Scenario:

1. Upon the update of a user (via the “update User” API)
2. Calculate the new resultant vector by adding T_n to T_{n-1}
3. Calculate the new state of the user affinity for T_n
4. If the T_{n-1} is different from the T_n state:
    a. Store the new state in a persistent store as the current state for the user

Exceptions:

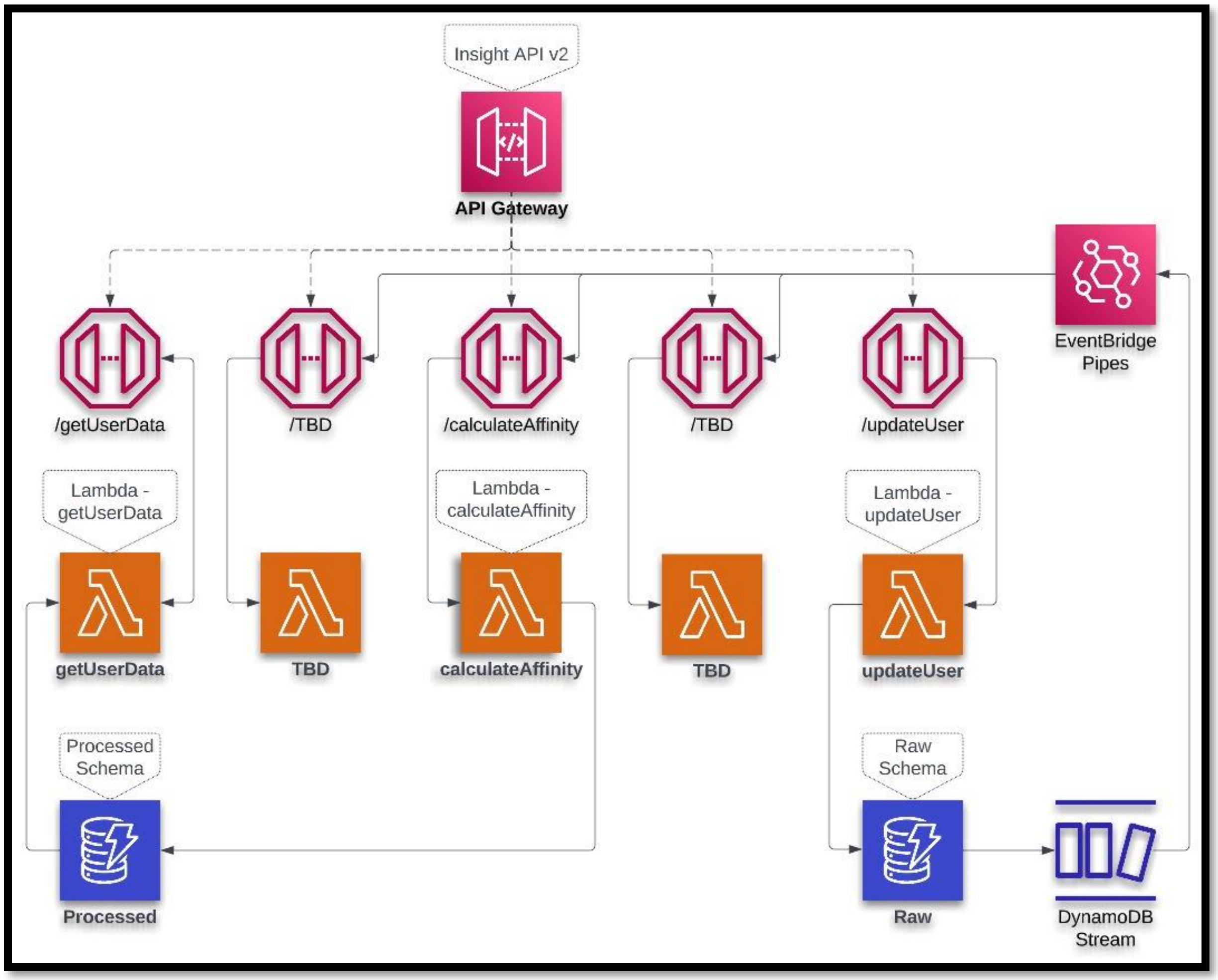
1. Unsuccessful operation should be fully logged with:
    a. User: <email>
    b. 2 x n matrix containing all attribute values for T_{n-1} and $T_{n\ to}$
    c. Timestamp of the failure
    d. Any system or OS related error message
2. Log the webhook call and the entire payload

Priority:	High
Channel to actor:	System / Change Data Capture
Frequency of use:	Up to many thousands of times per day.
Secondary actors:	
Channels to secondary actors:	

Open Issues:

4.2 Detailed Architecture

Our design works on a platform of integrated web services hosted by Amazon known as AWS services. These services are as follows: API Gateway, Lambda, DynamoDB, and EventBridge. The connection of these services is documented as such:

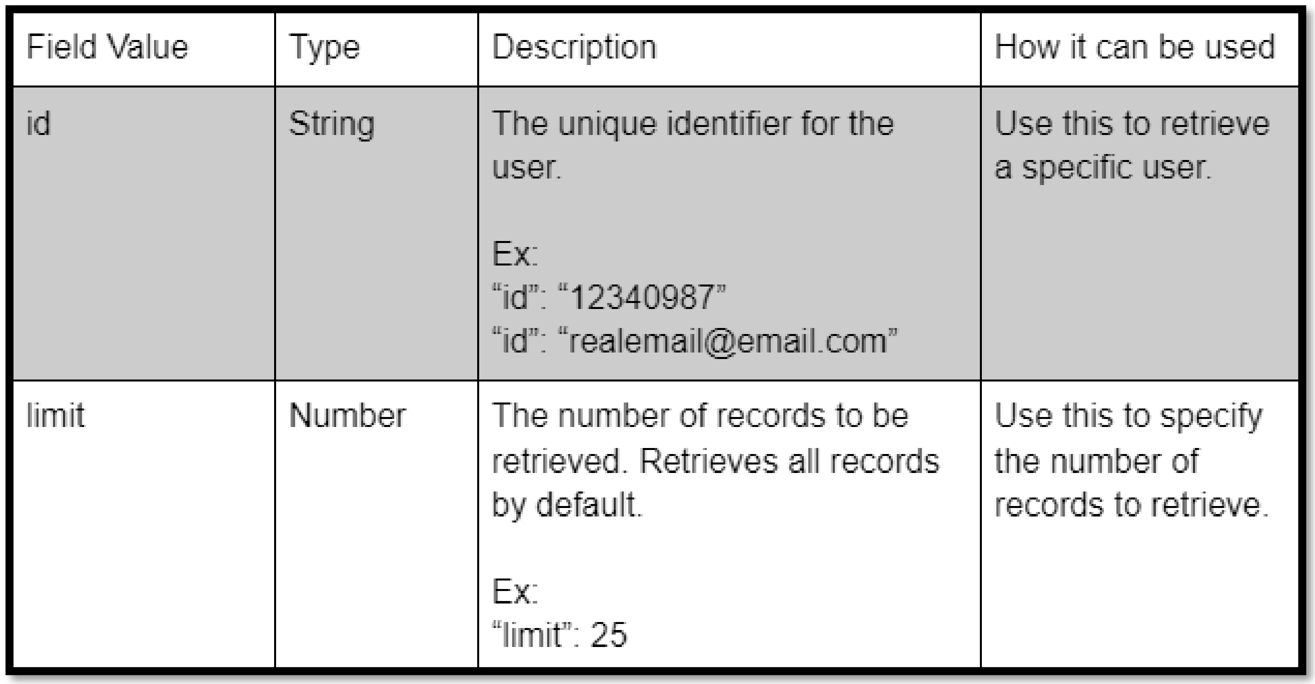


The client calls API Gateway to update and receive data records. The API getUserData endpoint will trigger the getUserData lambda function, while the updateUser endpoint will trigger the updateUser lambda function. Furthermore, the getUserData endpoint utilizes only the GET method, while the updateUser endpoint utilizes only the PUT method. This means getUserData will be called to retrieve records, and updateUser to update records. Our objective is to analyze the user data that comes in by doing calculations on it, so we make use of two DynamoDB database tables. One to hold the raw data of users and another to hold its calculated analysis. The calculations are done by the calculateAffinity lambda function which is invoked by a call to the calculateAffinity endpoint. This endpoint employs the POST method as we wish to create a new record every time new user data is submitted. The changes in raw user data are captured by a DynamoDB stream which is being watched by an EventBridge pipe. The pipe calls the calculateAffinity endpoint with the changed data as input. This data is processed by the calculateAffinity lambda function, and the analysis is stored in a database table for retrieval. The use of EventBridge pipes allows expansion through more lambda functions handling raw or processed data in the future with ease and this is demonstrated by the TBD endpoints and lambda functions in the flow diagram. Integral to our design is the scalability of the number of value propositions. This means our application can be used by clients other than Ponga for the purpose of analyzing their users. This scalability is implemented by the unique design of our lambda functions and will be discussed later. Our stretch goals include designing more said lambda functions and integrating a queueing system for the lambda functions to ensure no data is lost.

The AWS services flow directly from one to another; DynamoDB acting as the heart of our design while Lambda brings the services together through functions. The first step in the data flow is a call to API Gateway from the client. A Ponga user will perform some sort of action triggering the client to send a properly formatted call to the respective endpoint. The getUserData endpoint receives query string parameters while the updateUser endpoint receives a JSON payload. Each endpoint launches an associated AWS Lambda function. These functions process an event containing the JSON payload in the body. The getUserData function does not utilize the JSON payload in the body as it only requires query string parameters. These parameters are User and Limit. User is the email tied to a user's Ponga account while Limit is the number of records to retrieve. If a request is made without the Limit parameter, the client receives all records on that user in batches of 50. A request cannot be made without the User parameter. We have two maps we define in Insight documentation. The User map is the response from getUserData containing the number of records requested on a particular user. The Record map is the JSON payload request sent to updateUser by the client.

The second step is API Gateway triggering a lambda function depending on the call it receives. This lambda function will access a DynamoDB database table to either store or retrieve information. If storing data, a user record will be either updated or created in the Raw table. The change in this table is captured by DynamoDB as a data stream and piped directly to the calculateAffinity endpoint, triggering its respective lambda function. The calculateAffinity lambda function will perform calculations considering this new data and store the results in the Processed table in DynamoDB. If retrieving data, N user records will be returned to the client as a list of JSON documents in batches of 50 with pagination ID. The Processed table stores all historical data on a user that has been processed, while the Raw table holds just the up-to-date raw data of each user. This is because Ponga wishes to see a client's trend when retrieving records and that the calculateAffinity function only requires the running total of a user's data values to do calculations. The scalability of EventBridge pipes paired with Lambda and API Gateway is why we prefer this design. In the future, anyone could come in with no knowledge of the connected design and work on a new lambda function with the purpose of utilizing the data from either table to add further functionality after the initial launch. This design befits a growing application and AWS eliminates a lot of the groundwork.

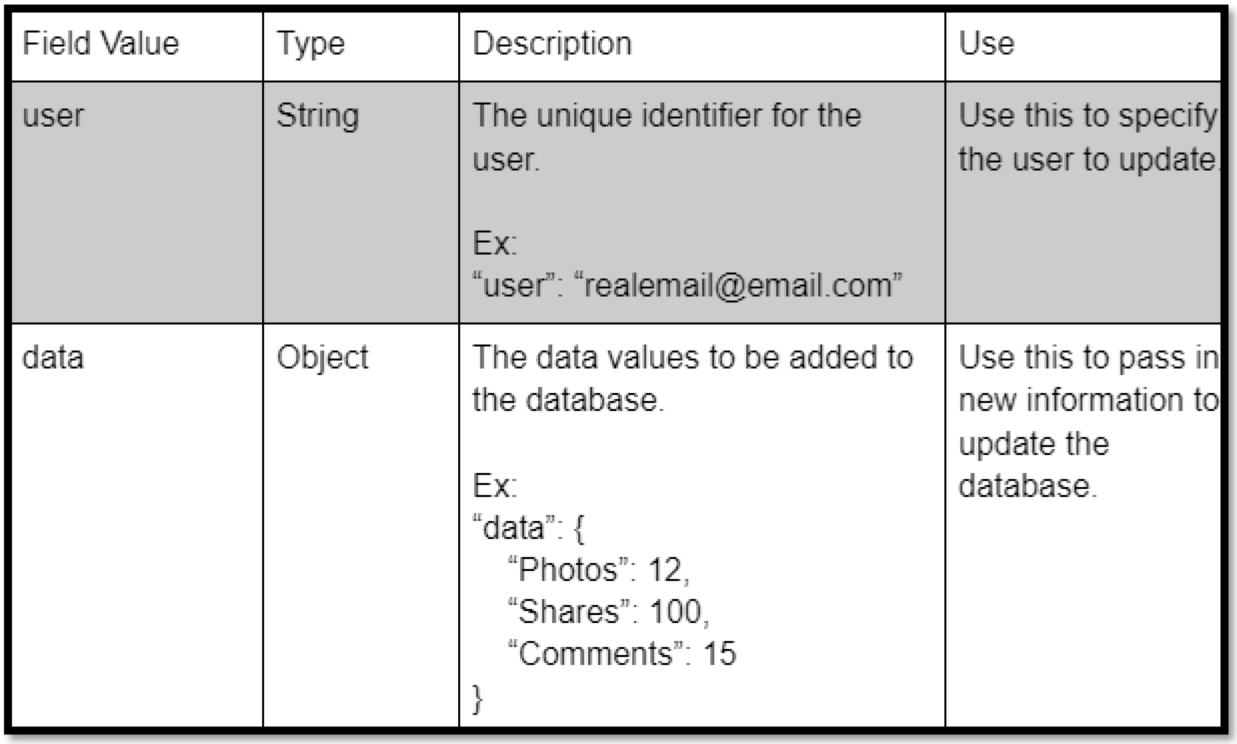
getUserData: This function receives an event payload from the API and returns data queried from the database. The event payload is expected to have the following fields:



Field Value	Type	Description	How it can be used
id	String	The unique identifier for the user. Ex: “id”: “12340987” “id”: “realemail@email.com”	Use this to retrieve a specific user.
limit	Number	The number of records to be retrieved. Retrieves all records by default. Ex: “limit”: 25	Use this to specify the number of records to retrieve.

The event payload is parsed for the ID and Limit fields. These are used in the query parameters to retrieve a certain number of records for a specific user. If no Limit field is given, then all data on a specific user will be returned in batches of 50, but this can be changed using the Limit field. The first query done by the function will retrieve the first page and keep querying until all requested records have been retrieved. This pagination is implemented in code using the LastEvaluatedKey map and the ExclusiveStartKey query parameter. The LastEvaluatedKey will be returned by the first query with the last key to be evaluated. This is used as the ExclusiveStartKey in the next query. If the request is successful, a response JSON payload will be created to be sent back to the client. The returned payload will include a status code, User map, and headers.

updateUser: This function receives an event payload from the API and updates the data in the database. The event payload is expected to have the following fields:

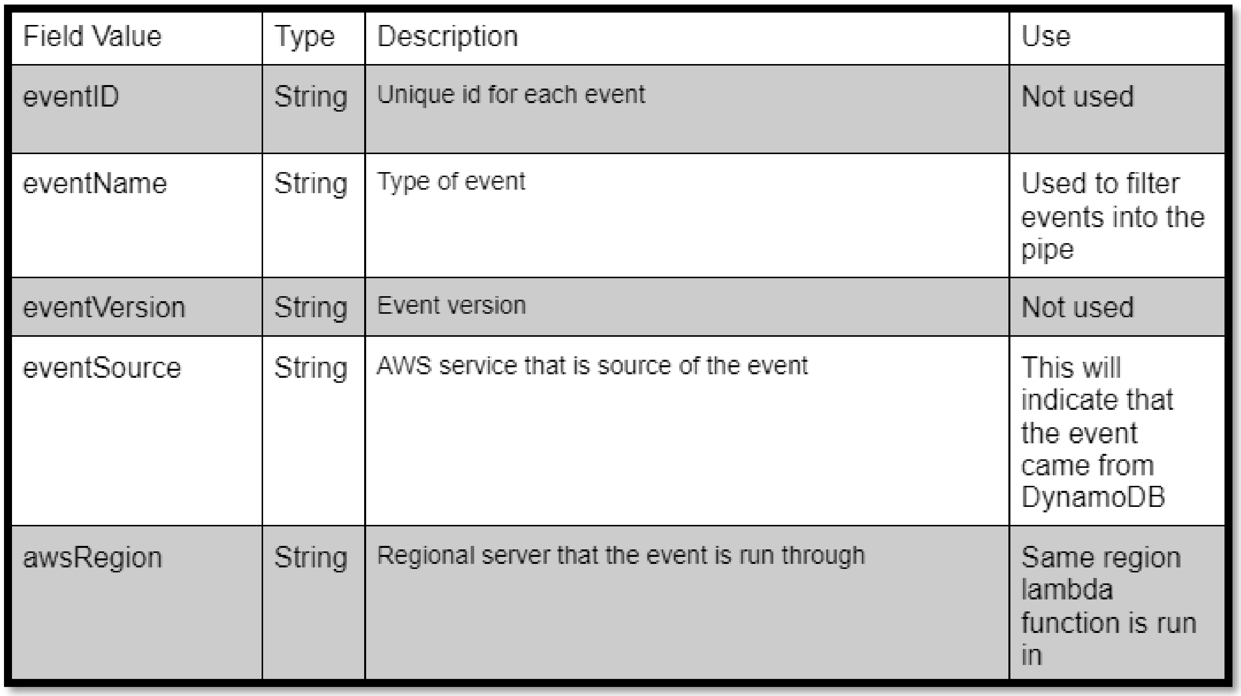


Field Value	Type	Description	Use
user	String	The unique identifier for the user. Ex: “user”: “realemail@email.com”	Use this to specify the user to update.

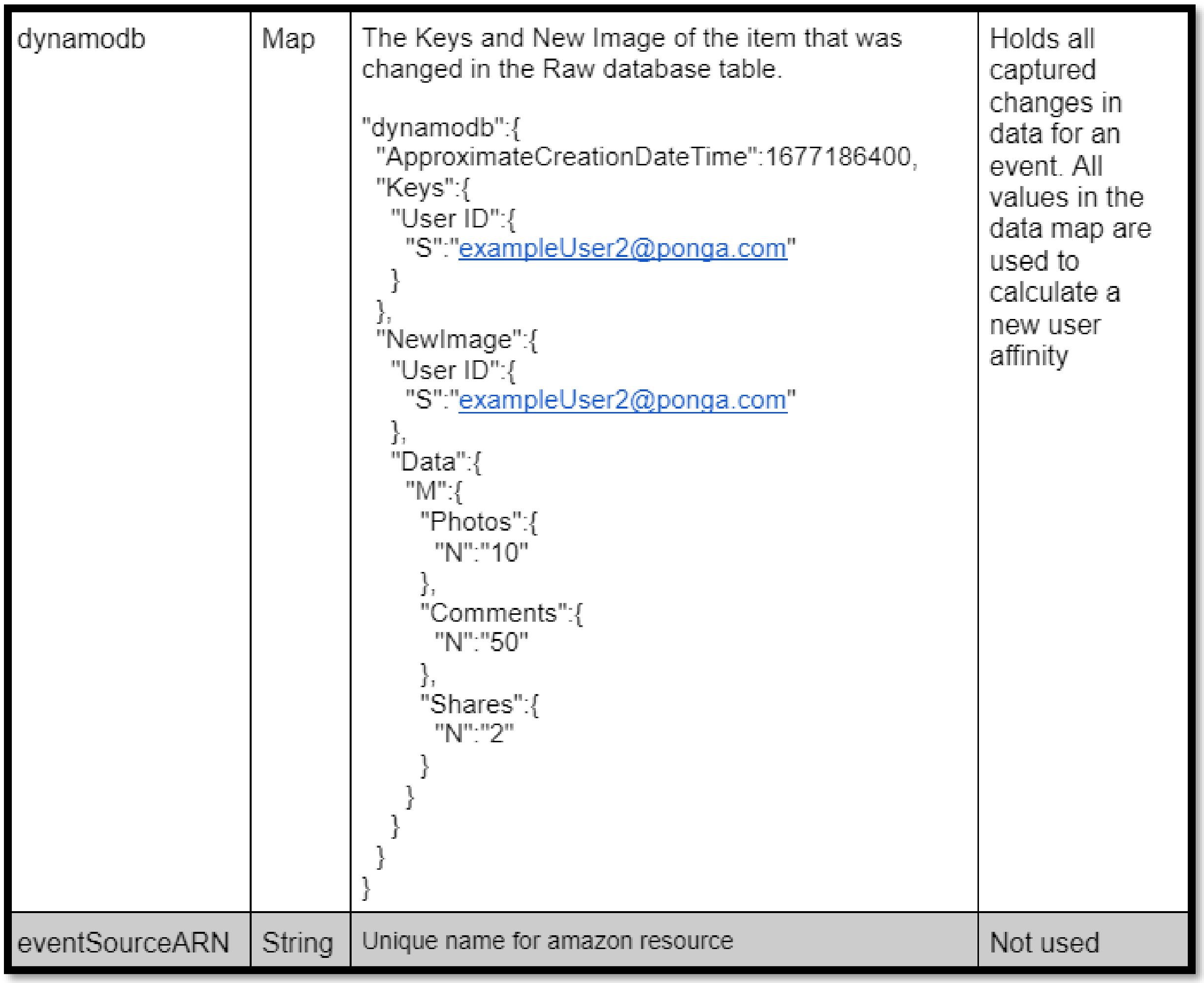
data	Object	The data values to be added to the database.
		
		Ex:
		“data”: {
		“Photos”: 12,
		“Shares”: 100,
		“Comments”: 15
		}
			Use this to pass in new information to update the database.

The event payload is parsed for the user and data fields. This function will then write the data as an update to an entry in the Raw table or create a new entry if the user parsed does not exist. This is done through a try catch that tries updating the record. If the record does not exist, the error will be caught and instead add the record. The scalability of value propositions comes into play in this function because we want to update the value propositions in data without knowing what they are beforehand. To explain how this is done, we must first describe the required parameters for an update query. These are Key, TableName, UpdateExpression, ExpressionAttributeNames, and ExpressionAttributeValues. The Key is the primary key (and sort key if there is one) of the table and TableName is self-explanatory. UpdateExpression is a string that contains clauses, attribute names, and their respective variables. These clauses are SET, REMOVE, ADD, and DELETE but we only use ADD and SET. Attribute names are Data, Timestamp, and whatever value propositions are in Data. These attributes are given variables that are defined by ExpressionAttributeValues. ExpressionAttributeNames is a map and only required when the names of your attributes are keywords in DynamoDB. Data and Timestamp are keywords, so they are only recognized by putting a hashtag in front of them in ExpressionAttributeNames. ExpressionAttributeValues is a map and defines the value to be added or set for each attribute. We want to take in data and format a query regardless of how many or what the value propositions are. Taking the length of the data object in the event body gives us how many value propositions there are. Then, using the object standard library of JavaScript, we can extract the names and values of each proposition. With this information we can fill out variables to be the maps for ExpressionAttributeNames and ExpressionAttributeValues as well as the string for UpdateExpression before performing the query. All a client has to do is send a data object through the API request body, and the function will extrapolate the information for the update query. A response JSON payload will be created to be sent back to the client. The response payload will include a status code and headers.

calculateAffinity: This function receives an event payload from EventBridge Pipe and queries the database to create an item in the Processed table for a specific user. The event payload is expected to have the following fields:



Field Value	Type	Description	Use
eventID	String	Unique id for each event	Not used
eventName	String	Type of event	Used to filter events into the pipe
eventVersion	String	Event version	Not used
eventSource	String	AWS service that is source of the event	This will indicate that the event came from DynamoDB
awsRegion	String	Regional server that the event is run through	Same region lambda function is run in



dynamodb	Map	The Keys and New Image of the item that was changed in the Raw database table.
		
		"dynamodb":{
		  "ApproximateCreationDateTime":1677186400,
		  "Keys":{
		    "User ID":{
		      "S":"exampleUser2@ponga.com"
		    }
		  },
		  "NewImage":{
		    "User ID":{
		      "S":"exampleUser2@ponga.com"
		    },
		    "Data":{
		      "M":{
		        "Photos":{
		          "N":"10"
		        },
		        "Comments":{
		          "N":"50"
		        },
		        "Shares":{
		          "N":"2"
		        }
		      }
		    }
		  }
		}
			Holds all captured changes in data for an event. All values in the data map are used to calculate a new user affinity

eventSourceARN	String	Unique name for amazon resource	Not used

The event payload body is parsed for the Data object and User ID string inside of the DynamoDB map. The number of value propositions is defined by the length of the Data object, and it is used to calculate the correct number of angles. If N is the number of value propositions, then the number of angles to be calculated would be (N(N-1))/2. An array of objects is used to store all calculated angles and each object has a Name, Value, and Dominant. The name defines which value propositions are being compared. For example, one name could be Photos vs Comments, and this would be called a relationship. Value holds the actual angle in degrees calculated using the inverse tangent of the two value propositions. Dominant is for client readability and states which of the value propositions is dominant in the relationship because reading the angle value alone is not clear. If there is no dominant proposition it will just state them as equivalent. This array of objects is stored in the Processed database table as the "Angles" attribute. The magnitude calculated is that of the value propositions and is stored in the Processed table as the "Magnitude" attribute. The function design allows for any number of value propositions and easy scalability for any client.

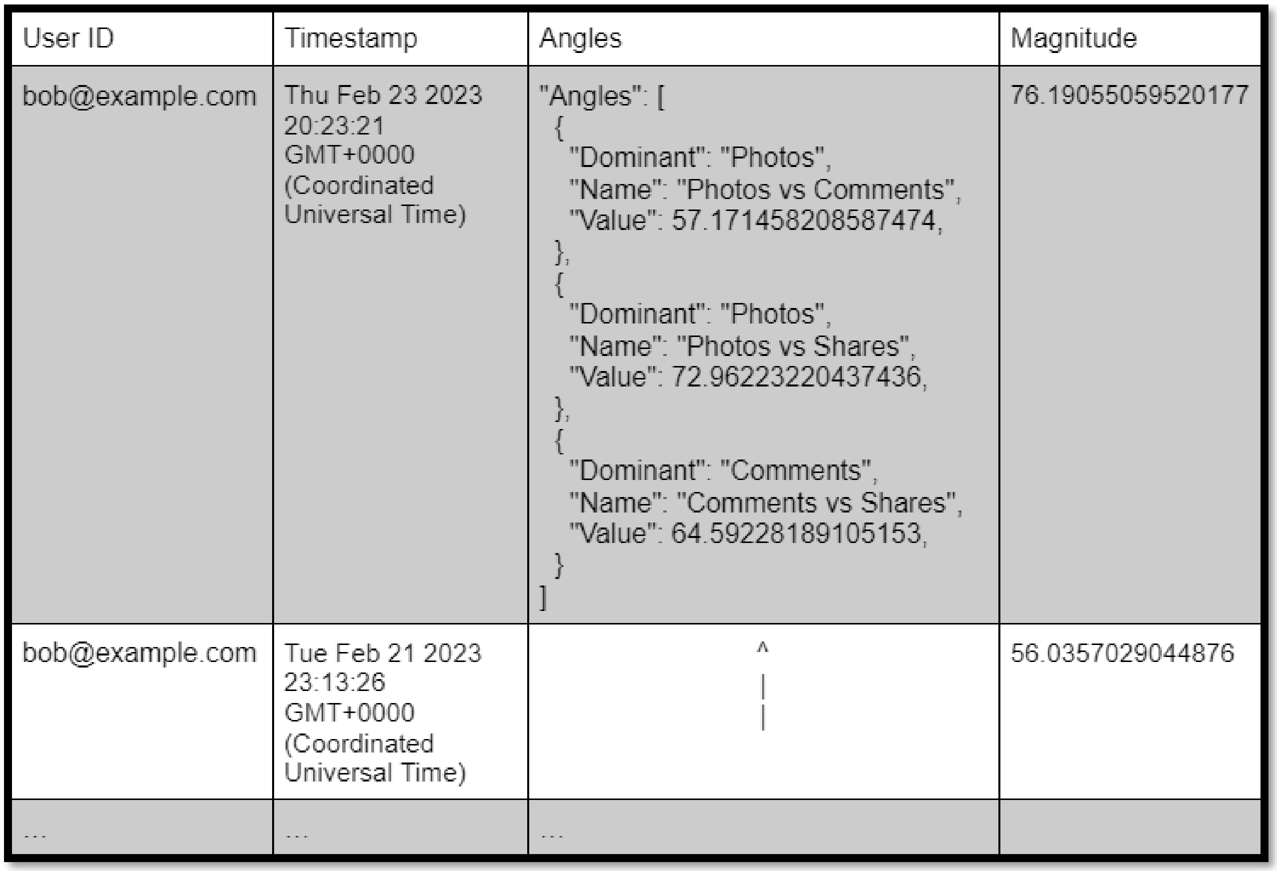
DynamoDB provides several options for a database such as: a sort key, capacity mode, indexes, encryption, point-in-time-recovery, time to live, and stream details. The sort key allows for multiple records on the same user. We use a sort key for the Processed table because one user has multiple data entries. There are two capacity modes known as on demand and provisioned. On-demand simply bills for the actual reads and writes an application performs. Provisioned allows you to optimize your costs by allocating read/write capacity in advance. We use on-demand as we do not have the subsequent knowledge required for provisioned. Tables in DynamoDB can utilize local and global indexes. We don’t use either form of index because our tables are only queried based on partition key. Amazon provides encryption for their databases, and we will be utilizing it. Point-in-time recovery is a feature that provides continuous backups of DynamoDB data. Time to live is a feature that can put a lifetime on specific attributes or data in your tables. We will use point-in-time recovery and time to live to safeguard the database records and keep them fresh. Stream details refers to a data stream containing changes in data. DynamoDB provides a stream but also allows the use of a Kinesis data stream. We use the stream given by DynamoDB and feed events into an EventBridge pipe.

The Processed table holds all historical data on each user that has been analyzed and ready for the client. As DynamoDB is a NoSQL database, the only columns of the table that are defined are the partition and sort key. Insight produces two attributes' "Angles" and "Magnitude" per record. Each user will gain a new record for each new action taken.

Partition Key: User ID - user's email

Sort Key: Timestamp - ISO 8601 date and time convention

Example table with multiple records for bob:



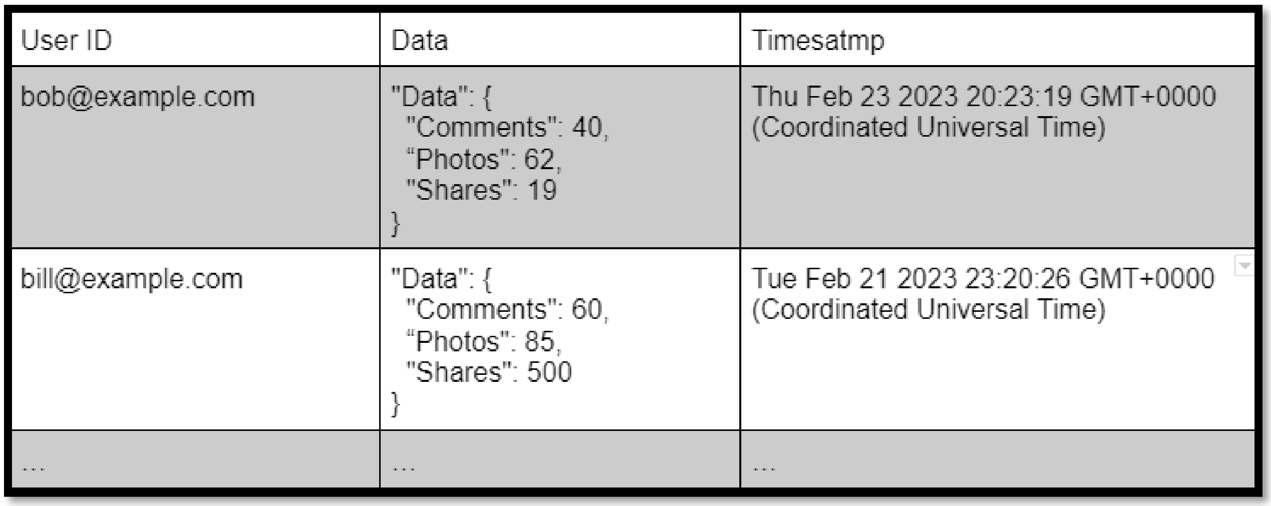
User ID	Timestamp	Angles	Magnitude
bob@example.com	Thu Feb 23 2023 20:23:21 GMT+0000 (Coordinated Universal Time)	"Angles": [{ "Dominant": "Photos", "Name": "Photos vs Comments", "Value": 57.171458208587474, }, { "Dominant": "Photos", "Name": "Photos vs Shares", "Value": 72.96223220437436, }, { "Dominant": "Comments", "Name": "Comments vs Shares", "Value": 64.59228189105153, }]	76.19055059520177
bob@example.com	Tue Feb 21 2023 23:13:26 GMT+0000 (Coordinated Universal Time)	^ | |	56.0357029044876
…	…	…	

The Raw table holds raw data on a user that is irrelevant to the client. As DynamoDB is a NoSQL database, the only columns of the table that are defined are the partition and sort key. Insight produces two attributes' "Data" and "Timestamp" per record. Each user will only have one record which updates with each new action taken.

Partition Key: User ID - user's email

Sort Key: None

Example table with one record for each user bob and bill:



User ID	Data	Timesatmp

bob@example.com	"Data": {
	"Comments": 40,
	"Photos": 62,
	"Shares": 19
	}
		Thu Feb 23 2023 20:23:19 GMT+0000 (Coordinated Universal Time)

bill@example.com	"Data": {
	"Comments": 60,
	"Photos": 85,
	"Shares": 500
	}
		Tue Feb 21 2023 23:20:26 GMT+0000 (Coordinated Universal Time)

…	…	…

To utilize Insight with a large-scale application like Ponga, a queue for incoming API requests will be required. This is a stretch goal planned for version 2 of the design. Also, any additional Lambda functions to be added will be a part of version 2. A queue is needed because of the risk that API Gateway goes down, causing any pending requests to come back empty, and there is no way to handle lost requests. To fix this, we would connect a queue to store all API requests until API Gateway is ready to accept a new request. This can be done through Amazon Simple Queue Service (SQS). All requests will be queued to ensure that they eventually make it to API Gateway. SQS would act as a middleman for API Gateway and the Ponga client. Taking advantage of AWS to implement our API will allow for near infinite scalability and easy modulation in the future.

4.3 Risks

Risk	Risk Reduction
Setting up database schema	DynamoDB is a Managed NoSQL Database that is optimized for performance at scale and can store more complicated objects such as JSON files. We will have a complete design of the database schema before creating it.
Some queued requests could become lost before triggering a Lambda function	Have input data stored in an amazon SQS queue. Then the API listener can draw from it.
Rigid Design - We might want to add new functionality other than just calculating user affinity in the future.	The use of EventBridge allows us to add many more Lambda functions that take the same event as calculateAffinity.

4.4 Tasks

1. Meet with sponsor to discuss the project.
2. Research 3D vector spaces and AWS (Amazon Web Services).
3. Understand AWS SQS, API Gateway, Lambda, DynamoDB, Kinesis – investigate API choices (http, rest, etc.), practice/learn programming lambda functions, investigate NoSQL and DynamoDB.
4. Understand the use cases given by sponsor. – Formulate questions for the sponsor as a team.
    a. Submit Updated Metric
    b. Retrieve Historical Data
    c. Calculate Attribute Affinity and Persist – Address mathematics of the function to be created, so we can divide up work on it in the future.
5. High Level Design of project
    a. API Gateway – Decide on API type and number of API being created
        i. GET method – Do we use query string parameters or take in a JSON payload from the client?
        ii. POST method – What should the response payload look like?

   b. Lambda – Decide number of functions needed and any additions for future versions. Plan to stub out each lambda function for testing.
      i. retrieve – Define event payloads. (Received and sent)
      ii. update – Define event payloads. (Received and sent)
      iii. calculateAffinity – Define event payloads. (Received and sent)
   c. DynamoDB – Decide number of tables and name them.
      i. raw table – Define rough schema.
      ii. processed table – Define rough schema
   d. Kinesis - Study its connection to the raw table and how it will send data to the calculateAffinity Lambda.
   e. SQS - Decide whether this will be part of version 1 of the project.
6. Fill out preliminary proposal and create presentation – Decide roles for the presentation and have everyone complete a section of the proposal.
7. Present High-Level Design to sponsor.
8. Decide on a Sprint planning method for next semester.
9. Prepare for a long meeting on 11/11 and decide on Sprint planning method.
10. Understand Ponga's use of AWS services and apply it to our own project.
11. Create contracts between the AWS services being used that serve as documentation on our project. Define what each block of the project requires and how the blocks communicate.
    a. API
       i. GET method – Document JSON map User. Display a sample request and response from the client.
       ii. POST method – Document JSON map Record. Display a sample request from the client.
    b. Lambda – Define input event payloads and functionality.
       i. retrieve
       ii. update
       iii. eventBridgePush – explain use of EventBridge

        iv. calculateAffinity
    c. DynamoDB – Define schemas.
        i. Raw
        ii. Processed

12. Complete preliminary report and create presentation – Decide roles for the presentation and have everyone complete a section of the report.
13. Setup AWS workspace for next semester using Cloud9.
14. Run AWS cost calculator.
15. Update task list and schedule.
16. Sprint 1 - Create DynamoDB database schemas and test them.
    a. Raw
    b. Processed
17. Sprint 2 – Stub out and test Lambda Functions and API methods.
    a. retrieve
    b. update
    c. eventBridgePush
    d. CalculateAffinity
    e. GET method
    f. POST method
18. Sprint 3 – Implement Lambda Functions and API methods
    a. retrieve
    b. update
    c. calculateAffinity
    d. GET method
    e. POST method
19. Sprint 4 – Prepare all blocks of Insight for delivery and testing.
    a. Update documentation with final design decisions and more specifics on code.
    b. Create account for testing Insight.
    c. Fill tables with data for testing use cases.

d. Create step by step guide to AWS once logged in with sample API calls.

4.5 Schedule

Tasks	Dates	Details	Contributor
Meet with sponsor	9/29	We will meet in JBHT 532 and use the whiteboard for a detailed description of the sponsor's desired design.	All
Research	9/30-10/6	For this week, the team will each research AWS concepts and how they connect to our project.	All
Understand AWS services connection	10/7	Zoom call to discuss the use of each AWS service after having researched them.	All
Understand the use cases given by sponsor	10/8-10/14	Discuss use cases as a team and formulate questions for the sponsor. Zoom call at the end of the week to address questions.	All
High Level Design	10/15-10/23	Meet in discord several days to complete proposal and presentation.	All
i. DynamoDB	10/15	Decide on the number of tables. Discuss schema ideas for our DynamoDB tables.	
ii. Kinesis and SQS	10/18	Understand how kinesis will work for our project. Decide on SQS being part of version 1.	
iii. API Gateway and Lambda	10/21	Decide on API type and number. Make design decisions for the GET and POST endpoints.	
Proposal and Presentation	10/15-10/23	Complete required sections of the proposal while defining High Level Design. Create and practice presentation at the end of the week.	All

Present High-Level Design to sponsor	10/28	Zoom call with sponsor. Present current design to sponsor and get feedback. Discuss sprint planning.	All
Sprint planning	10/29-11/4	Research Sprint planning methods and applications for organization. Decide on one for next semester. 20-minute zoom call on 11/4.	All
Review and prepare	11/5-11/10	Prepare for meeting on 11/11 and decide on Sprint planning method. Look over proposal comments and fix errors.	All
Understand Ponga's AWS system	11/11	3 hours zoom call diving into Ponga's use of AWS and how to apply it to our project.	All
Documentation	11/14-11/26	Document all aspects of the Project and form Sprints around the documentation.	All
i. API Gateway	11/14-11/15	Document the requests and response of each endpoint. Formatting the payloads that the client sends and receives.	Matthew
ii. getUserData and updateUser lambda	11/18-11/19	Document the lambda functions to specify input events and formatting of output event.	Ryan
iii. eventBridgePush and calculateAffinity lambda	11/25-11/26	Document conversion of data stream to event, and input/output event of calculation lambda.	Julio Logan
iv. DynamoDB tables	11/17	Document database table schemas and connection to design.	
Report and Presentation	11/26-11/28	Complete required sections of the report while defining detailed design and next semester sprint plans. Create and practice presentation at the end of the week.	All
Setup AWS workspace	11/29-12/8	Finalize AWS accounts and setup Cloud9	Matthew

Run AWS cost calculator	1/18	Estimate cost of creating and testing the Insight Project	Matthew
Update task list and schedule	1/19	Change Sprint structure based on sponsor input and add a few tasks before starting sprint 1	Ryan
Sprint 1 – DynamoDB Implementation	1/20-1/22	Create DynamoDB database schemas	Matthew
i. Raw	1/22	Test DynamoDB streams with CloudWatch and eventBridgePush	Julio Dylan Logan
ii. Processed		Test once calculateAffinity has been stubbed out	
Sprint 2 – Stub Lambda and API	1/23-2/5	Stub out and test Lambda Functions and API methods	All
i. Retrieve	1/23	Stub get user data lambda (Can return records from the processed table to client through an API call)	Matthew
ii. Update	1/24	Stub update user lambda (Can create new or update existing records in the raw table through an API call from the client)	Ryan
iii. Event Bridge Push	1/27-2/2	Stub EventBridge push lambda (Ended up scrapping EventBridge push lambda function and using EventBridge Pipes instead. These push changes in the raw table to calculate affinity lambda.)	Julio Logan
iv. Calculate Affinity	2/3-2/5	Stub calculate affinity lambda (Takes in event from the EventBridge Pipe, then puts an item into the processed table. Calculation design will be in the implementation phase. We need to flush out a way of reading events in the lambda as well.)	Dylan Logan

v. GET	1/23	Stub GET method in API Gateway (created automatically when generating Get User Data lambda)	Matthew
vi. POST	1/23	Stub POST method in API Gateway (created automatically when generating Update User Lambda)	Matthew
Sprint 3 – Implement Lambda and API	2/6-3/19	Implement Lambda Functions and API methods	All
i. Retrieve	2/20-3/03	Implement getUserData (Add exception error codes and test pagination id when requesting records)	Julio Matthew
ii. Update	2/20-2/24	Implement updateUser (implement the adding of data to existing records rather than replacing the existing records. Also add exception error codes.)	Julio Logan
iii. Calculate Affinity	2/13-2/17	Implement calculateAffinity (code for calculations and the scaling of value propositions for use by any company, not just Ponga)	Matthew Ryan Dylan
iv. API	2/6-2/7	Clean up methods in API Gateway (modifying endpoint methods to look more professional and presentable)	Matthew
Sprint 4 – Prepare deliverables	3/20-4/23	Create an AWS testing account, tables, and guide to allow testing on individual functions and flows.	All
i. Update documentation	3/20-4/2	Update Final Report and Insight Documentation	Dylan Logan
ii. Setup testing account	4/3-4/9	Set up a testing account with all functions deployed	Matthew
iii. Pre-populate tables	4/3-4/9	Fill tables with sufficient data to test use cases.	Ryan Julio
iv. Create guide to test	4/10-4/23	Steps guiding tester on the functionality of Insight on AWS	Matthew

			Ryan Juilo

4.6 Deliverables

- Design Documentation: Google docs linked to a Lucid chart. These contain documentation on each major block of Insight.
    1. For API this would be sample requests and responses when applicable.
    2. For Lambdas, we will provide input event payloads and functionality.
    3. For database tables, we provide the schemas.
- Login credentials to AWS account where all API, Lambdas, and DynamoDB tables have been deployed.
- Step by Step guide to testing Insight utilizing CloudWatch logs once logged into AWS with full access.
    1. Pre-populated data in the Raw and Processed tables upon login.
    2. Tour of AWS blocks used for Insight.
    3. Series of API calls to test all use cases.
- API and Database Response Times: Timing data related to each APIs response times in connection with the database.
- Final Report: Contains background information, use cases, detailed project and database design, tasks, schedule, deliverables, key personnel, and references.

5.0 Key Personnel

Dylan Vaughn – Vaughn is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed Intro to Engineering. No relevant experience or internships. Was responsible for research and understanding AWS.

Matthew Clemence – Clemence is a senior Computer Engineering major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed Software Engineering and Intro to Engineering. Was responsible for hosting weekly meetings and research.

Ryan Drake – Drake is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed Database Management Systems, Software Engineering, and Algorithms courses. Was responsible for researching and understanding AWS and the services we will be using.

Julio Bonilla – Bonilla is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed Database

Management Systems, Algorithms, Software Engineering, Computer Networks, and Intro to Engineering. No relevant experience or internships. Was responsible for outlining the objective, research, unit testing, and coordinating team meetings.

Logan Reed – Reed is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed Database Management Systems, Big Data Management and Analyzation, and Software Engineering. Has experience in database and API relations in personal projects. No relevant internships. Was responsible for outlining, researching and understanding DB relations.

Jerrel Fielder – Industry Champion. Senior technology and business executive with 30 years in Enterprise and Consumer software in multiple roles: technology, technology leadership, architecture, consulting, sales, sales leadership, and founder.

6.0 Facilities and Equipment

The utilities we will make use of are all the Amazon Web Services. The platform we are building will employ API Gateway, Lambda, DynamoDB, Kinesis, and EventBridge. These services will seamlessly flow together to accomplish the implementation of our solution.

API Gateway accepts requests to access the API and provides the promised services. API must be formatted correctly. Amazon Lambda is a dynamic service which can accomplish many tasks using programmable functions, and it allows for each service to be easily connected. DynamoDB is the NoSQL database that we will be using to store and access data. Kinesis will be used in conjunction with DynamoDB to detect changes in the tables so that certain functions can be called in response. EventBridge will handle the bus of events passed from Kinesis change data capture to Lambdas.

7.0 References

[1] Amazon. “What Is AWS Lambda? - AWS Lambda.” *Amazon Web Services*, 1 July 2022, docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[2] Amazon. “What Is Amazon DynamoDB? - Amazon DynamoDB.” *Amazon Web Services*, 15 June 2022, docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

[3] Amazon. “Welcome - Amazon API Gateway.” *Amazon Web Services,* 14 Nov. 2022, docs.aws.amazon.com/apigateway/latest/api/Welcome.html.

[4] Amazon. “What Is Amazon Kinesis Data Streams? - Amazon Kinesis Data Streams.” *Amazon Web Services*, 29 Nov. 2021, docs.aws.amazon.com/streams/latest/dev/introduction.html.

[5] Amazon. “What Is Amazon EventBridge? - Amazon EventBridge.” *Amazon Web Services*, 14 Nov. 2022, docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html.

[6] Amazon. “What Is Amazon Simple Queue Service? - Amazon Simple Queue Service.” *Amazon Web Services*, 20 Oct. 2022, docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html.

[7] Google. “How Google Analytics Works - Analytics Help.” *Google Help*, 2022, https://support.google.com/analytics/answer/12159447?hl=en.

[8] Vanhee, David. “Google Analytics 4 Click Tracking – The Beginners’ Guide.” *Data Driven*, Digital Mantis, Inc, 19 Aug. 2022, https://www.datadrivenu.com/how-to-track-clicks-in-googleanalytics-4/. Accessed 25 Nov. 2022.

[9] “Scroll Depth Trigger - Tag Manager Help.” *Google Help*, Google, 2022, https://support.google.com/tagmanager/answer/7679218?hl=en.

[10] “Recordings: The Complete Guide.” *What Are Session Recordings (Session Replays) + How to Use Them*, Hotjar, 6 June 2022, https://www.hotjar.com/session-recordings/.