Cloud and Big Data Final Report — Columbia Coffee Chat

Chun-Hua Lin
Columbia university
cl4335@columbia.edu

Shawn Tai
Columbia university
ht2539@columbia.edu

Daniel Guo
Columbia university
dg3287@columbia.edu

Luke Hsu
Columbia university
th2881@columbia.edu

***Abstract*—This paper proposes the development of a coffee chat platform specifically designed for Columbia University Master's students to enhance networking opportunities and foster connections among the student community. The platform will utilize tools such as Amazon Web Services' Lambda, Cognito, DynamoDB, S3, CloudWatch Scheduler, and SES, as well as the agility of React to provide a simple and user-friendly solution. Existing networking applications have limitations that our platform aims to overcome by catering to students, providing ease of use, and offering a wider range of interest-based connections. The platform will feature authentication, user accounts, and a matching algorithm to pair compatible students each week, with stretch goals of cafe suggestions, larger group coffee chats, chat functionality, and student reviews. By utilizing student data such as majors, classes, and interests, the platform aims to facilitate successful matches and expand to a broader consumer base in the future.**

***Index Terms*—Amazon Web Services, Lambda, Cognito, DynamoDB, S3, CloudWatch, CodePipeline, CodeBuild, SES, CloudFormation, React**

I. Introduction

Building a sense of community among Master's students at Columbia University poses a significant challenge due to limited networking opportunities and social events. To address this issue, we propose the development of a coffee chat platform specifically designed for Columbia students. Our platform aims to enhance networking opportunities and foster connections among the student community, filling the void that currently exists in this space.

Networking is crucial for students preparing to enter the workforce, with 85% of jobs filled through networking according to HubSpot. Unfortunately, Columbia's Master's students miss out on the opportunity to build a strong network of highly skilled engineers due to the lack of events catered specifically to their needs. Our platform seeks to provide a simple and user-friendly solution, enabling students to expand their professional network through coffee chats.

While existing applications such as Randomcoffee, Meetup, and Atleto offer similar solutions, they have limitations that our platform aims to overcome. Randomcoffee primarily targets corporate settings and is not accessible to all consumers. Meetup requires constant user involvement in joining groups and finding events. Atleto focuses on connecting individuals based on sports interests only. Our platform improves upon these limitations by catering to students, providing ease of use, and offering a wider range of interest-based connections.

To ensure the success of our platform, we have set specific targets and success criteria. Columbia students will be able to sign up for weekly coffee chats with new students who share similar interests. The platform will feature authentication, user accounts, and a matching algorithm to pair compatible students each week. Stretch goals include cafe suggestions, larger group coffee chats, chat functionality, and student reviews to enhance matching algorithms.

Our platform initially targets Columbia students, but we envision expanding to a broader consumer base in the future. By utilizing student data such as majors, classes, and interests, we aim to facilitate successful matches. Furthermore, integrating cafe data by location could enhance the meeting experience. Our goal is to provide a robust platform that can handle periodic high traffic without wasting resources during low-traffic periods.

II. Solution Architecture

Our platform is built utilizing the benefits and agility of React and robust functions provided by Amazon Web Services (AWS).

React is a flexible JavaScript library for building user interfaces, known for its agility in allowing developers to build and maintain complex UIs quickly and efficiently by breaking them down into reusable components. React allows developers to build robust, scalable, and responsive applications that meet the specific needs of their users and business requirements.

Amazon Web Services (AWS) is a cloud computing platform that provides a wide range of infrastructure and services to businesses and individuals worldwide. It's designed to provide businesses with an easy, cost-effective way to deploy and manage their applications and services in the cloud. It's built on a global network of data centers that allow customers to quickly deploy their applications and services in any region around the world with a pay-as-you-go model that enables customers to pay only for the resources they use.

A. Lambda

AWS Lambda is a serverless computing service that allows us to run code without provisioning or managing servers. Lambda can be used to build custom APIs, and back-end services, or to process data from other AWS services in multiple programming languages, including Node.js, Python, Java, and more.

B. Cognito

Amazon Cognito is a fully managed identity service that allows us to add user sign-up, sign-in, and access control to your web and mobile apps. It provides user authentication, user data synchronization, and access control features across multiple devices and platforms.

C. DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB is designed to handle any scale of application traffic while maintaining high performance and availability.

D. CloudWatch Scheduler

AWS CloudWatch Scheduler is a service that lets us schedule and automate AWS resources. With CloudWatch Scheduler, we can perform tasks at specific times and automate recurring tasks without having to manually manage and monitor them.

E. S3

Amazon S3 is a highly scalable, durable, and secure object storage service that allows you to store and retrieve any amount of data from anywhere on the web. With S3, we can store and retrieve data from a simple web interface, command-line interface, or programmatically via APIs.

F. API Gateway

Amazon API Gateway is a fully managed service that allows us to create, publish, maintain, monitor, and secure APIs at any scale. It enables developers to create RESTful APIs that act as a front door for applications to access data and functionality from back-end services, such as AWS Lambda, Amazon DynamoDB.

G. CloudFormation

Amazon CloudFormation is a service that allows you to define and manage infrastructure as code. It uses templates, which are essentially JSON or YAML files, to automate the creation of AWS resources such as EC2 instances, S3 buckets, Lambda functions, and more. It also provides the ability to rollback changes if they result in an error, and to track and audit changes made to your infrastructure.

H. CodePipeline

Amazon CodePipeline is a fully managed continuous delivery service that makes it easy to automate the release process for applications. It allows us to automate the steps required to release their code changes, from building and testing to deploying and releasing to production.

I. Simple Email Service

Amazon SES (Simple Email Service) is a cloud-based email service that provides a cost-effective way to send and receive email.

III. Software Design

1) Frontend: We chose React as the framework to develop our front-end application with. Some of its advantages include:

- **Component-based architecture:** React.js follows a component-based approach, which allows us to break our user interface into reusable and independent components. This makes it easier to manage and maintain our codebase, as well as promote reusability and modularity.
- **Virtual DOM:** React.js uses a virtual DOM (Document Object Model) to efficiently update and render components. Instead of manipulating the actual DOM directly, React creates a lightweight copy of it in memory, allowing for faster updates and better performance.
- **Efficient rendering:** React.js implements a diffing algorithm that only updates the necessary parts of the DOM when changes occur. This means that React intelligently determines what parts of the user interface need to be updated, minimizing the number of actual DOM manipulations and improving performance.

In terms of rendering efficiency, React provides hooks that make maintaining states and re-rendering the DOM a lot easier. Specifically, these are the two hooks we utilize in our application:

- **useState:** Managing State in Functional Components
  - useState is a hook in React that allows us to manage states within functional components. State refers to the data that can change over time and affects how your component behaves and renders.
  - Advantages of useState:
    - Simplicity: It simplifies state management by providing a straightforward way to create and update state variables.
    - Reactivity: When the state value changes, React efficiently re-renders only the parts of our component affected by that state, optimizing performance.
    - Multiple State Variables: We can use useState multiple times in a single component to manage different state variables independently.
- **useEffect:** Handling Side Effects in Functional Components
  - useEffect is another hook in React that allows us to perform side effects in functional components. Side effects are actions that are not directly related to rendering the component, such as fetching data from an API (which is the majority of our use cases in this app), subscribing to events, or manipulating the DOM.
  - Advantages of useEffect:
    - Lifecycle Management: useEffect enables you to handle different lifecycle events of a component, such as when it first mounts, updates, or unmounts.
    - Side Effect Dependencies: We can specify dependencies for useEffect to control when it should

be triggered. This ensures that the side effect runs only when the relevant data or state changes, preventing unnecessary re-execution.

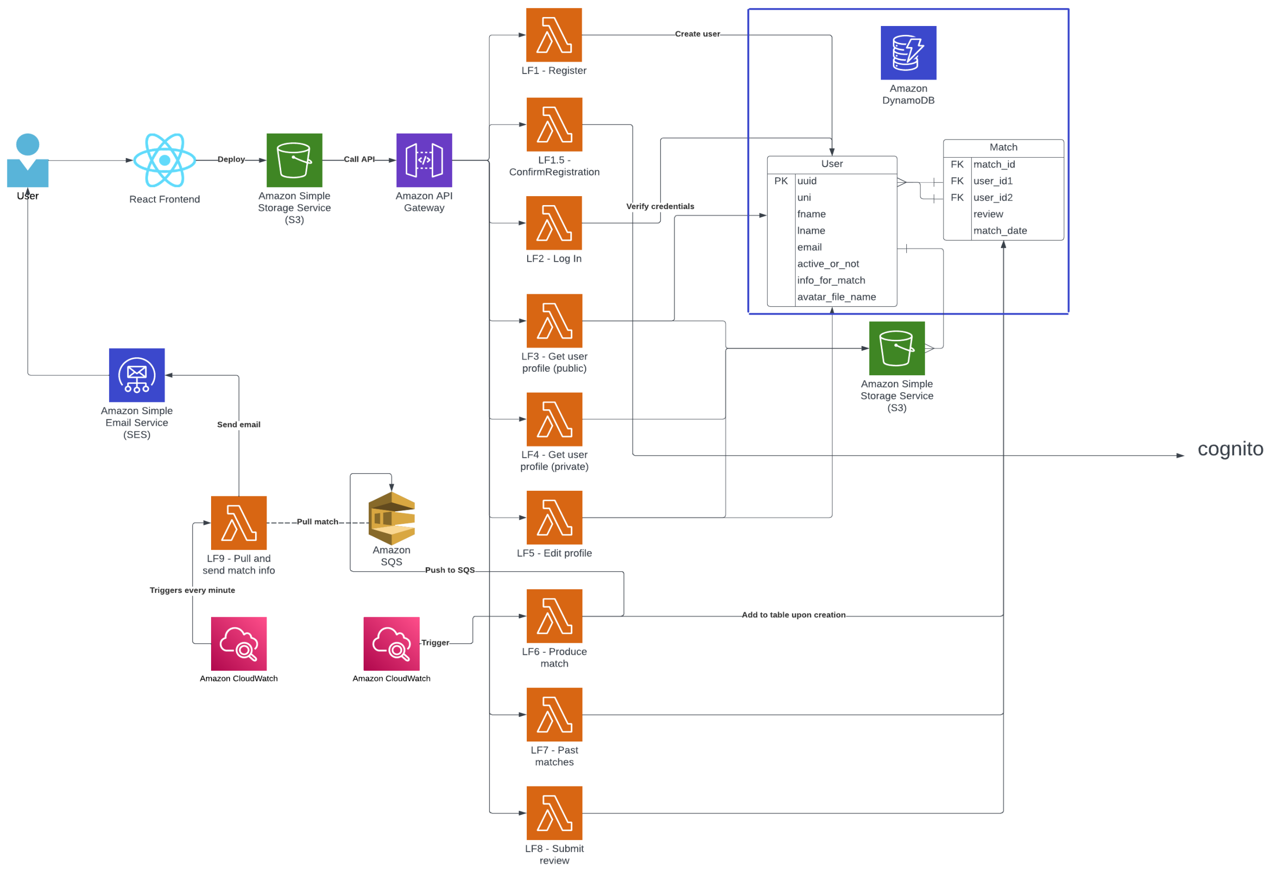
A. Backend

1) Website Hosting: we utilize a serverless frontend CI/CD pipeline to automate the deployment of our React frontend app using AWS *CloudFormation*, *CodePipeline*, and *CodeBuild*. The pipeline pulls the app's source code from our GitHub repository, builds it, and deploys it to an S3 bucket. The chosen AWS services offer several advantages, including simplified infrastructure management, seamless integration with other AWS services, version control and traceability, and continuous integration and deployment capabilities. Our CloudFormation template generates the necessary AWS resources, including S3 buckets, IAM roles, CodeBuild projects, and a CodePipeline pipeline. It defines the infrastructure as code, ensuring consistent and reproducible deployments across different environments. Here's a brief overview of our CI/CD pipeline:

1) Create an AWS CloudFormation stack using the provided CloudFormation template. This template defines the necessary AWS resources for the CI/CD pipeline, including S3 buckets, IAM roles, CodeBuild projects, and a CodePipeline pipeline.
2) In the CloudFormation parameters, provide the following information:
    a) GitHubRepo: The name of your GitHub repository containing the React app.
    b) GitHubBranch: The branch of the GitHub repository to use for deployment.
    c) GitHubToken: An access token with appropriate permissions to access your GitHub repository.
    d) GitHubUser: Your GitHub username.
3) Once the CloudFormation stack is created, it will set up the required AWS resources.
4) The CodePipeline pipeline consists of three stages: Source, Build, and Deploy.
    a) In the Source stage, the pipeline is configured to pull the React app source code from your specified GitHub repository and branch.
    b) In the Build stage, CodeBuild is used to build the React app. The ClientBuildProject CodeBuild project is responsible for installing the necessary packages and running the build script. The build artifacts are stored in the pipeline.
    c) In the Deploy stage, the DeployClientBuildProject CodeBuild project deploys the built React app to an S3 bucket. The app files are copied to the ClientBucket S3 bucket using the AWS CLI.
5) Once the pipeline is set up, it will automatically trigger whenever changes are pushed to the specified GitHub repository and branch.
6) The deployed React app will be accessible through the URL provided in the CloudFormation stack outputs. The WebsiteURL output provides the URL for the website hosted on the S3 bucket.

2) API: The APIs are first definied using *Swagger*, which allows us to define our APIs to generate interactive documentation, client libraries, and server stubs in a variety of programming languages. Then, the APIs are deployed on *API Gateway* to connect the functionalities to the frontend. *API Gateway* also provides a CORS feature that can be enabled for a REST API. Cross-Origin Resource Sharing (CORS) is a security feature implemented in web browsers that restricts web pages from making requests to a different domain than the one that served the original web page.

- **login:**
    - Method: Post
    - Request:
        * ”username_or_email”: ”string”,
        * ”password”: ”string”
    - Respond:
        * ”status”: true,
        * ”authToken”: ”string”
    - Description: The login API call is directed to the *lf2-login* Lambda function, which leverages AWS Cognito to authenticate the user's login credentials and generate a JSON Web Token (JWT) for future user verification.
- **register:**
    - Method: Post
    - Request:
        * ”username”: ”string”,
        * ”email”: ”string”,
        * ”password”: ”string”
    - Respond:
        * ”status”: true
    - Description: The register API call is handled by the *lf1-register* Lambda function, which utilizes AWS Cognito to add the user's information to the user pool.
- **confirm:**
    - Method: Post
    - Request:
        * ”username”: ”string”,
        * ”confirmation_code”: ”string”
    - Respond:
        * ”status”: true
    - Description: The confirm API call is handled by the *lf1dot5-ConfirmRegistration* Lambda function, which utilizes AWS Cognito to verify the user's email information by validating the confirmation code received in their email.
- **/profile/public/{userid}**:
    - Method: Get
    - Respond:
        * ”profile”: { *user_info...* }
    - Description: The get public profile API call is handled by the *lf3-get-user-profile-public* Lambda



User	
PK	uuid
	uni
	fname
	lname
	email
	active_or_not
	info_for_match
	avatar_file_name

Match	
FK	match_id
FK	user_id1
FK	user_id2
	review
	match_date

*0.5

Fig. 1. An overview of our software architecture.

function, which retrieves the user's public profile information and profile photo from DynamoDB and S3 bucket respectively, using their user ID passed in as a path parameter.

- **/profile/private:**
  - Method: Get
  - Respond:
    - ∗ "profile": { *user_info...* }
  - Description: The get private profile API call is handled by the *lf4-get-user-profile-private* Lambda function, which retrieves the user's private profile information and profile photo from DynamoDB and S3 bucket respectively, using decoded JWT token information which includes the user ID. This ensures that the user'sprivate profile only can be viewed with the verified user with the credentials.
- **/profile/edit:**
  - Method: Put
  - Request:
    - ∗ "profile": { *user_info...* }
  - Respond:
    - ∗ "profile": { *user_info...* }
  - Description: The edit profile API call is a POST request directed to the *lf5-edit-profile* lambda function. This function is responsible for updating user information and profile photo in the DynamoDB and S3 bucket, respectively. The user ID for the profile being edited is provided in the request body, which is used to identify the user to update. This API is restricted to the private profile page, ensuring that only verified users can make edits.
- **/match/current/{userid}:**
  - Method: Get
  - Respond:
    - ∗ "matchId": "string",
    - ∗ "user1Id": "string",
    - ∗ "user2Id": "string",
    - ∗ "review": "string"
  - Description: The get past match API call is directed to the *lf7-past-matches* Lambda function, which is responsible for retrieving the past matches information of the authenticated user. The function utilizes the DynamoDB SDK to query the database and

retrieve the match history based on the user ID extracted from the decoded JWT token. This ensures that only the authenticated user with valid credentials can view their own match history.

- **/match/past/{userid}:**
  - Method: Get
  - Respond:
    - ∗ ”matchId”: ”string”,
    - ∗ ”user1Id”: ”string”,
    - ∗ ”user2Id”: ”string”,
    - ∗ ”review”: ”string”
  - Description: The get current match API call is directed to the *lf7-past-matches* Lambda function, which is responsible for retrieving the current matches information of the authenticated user. The function utilizes the DynamoDB SDK to query the database and retrieve the match history based on the user ID extracted from the decoded JWT token. This ensures that only the authenticated user with valid credentials can view their own match history.
- **/match/submitReview:**
  - Method: Put
  - Request:
    - ∗ ”matchId”: ”string”,
    - ∗ ”review”: ”string”,
  - Respond:
    - ∗ ”status”: true
  - Description: The submit match review API call is directed to the *lf8-submit-review* lambda function, which uses the DynamoDB SDK to retrieve the past match’s review based on the match ID. This API allows users to submit reviews for their past matches.

3) Loosely Coupled Microservice: We implemented our microservices using AWS Lambda, which allowed us to build and deploy individual functions quickly and easily without the need to manage the underlying infrastructure. Each Lambda function is triggered by an API Gateway endpoint, and they are designed to be stateless, meaning that they don’t hold any session information. This approach provides scalability, and flexibility.

- **Lambda:** We have chosen Python as our preferred language for developing our microservices and are utilizing the Python SDK for AWS resources. This decision was made based on the familiarity and ease of use of the language, as well as the vast community support and available libraries for AWS services. By leveraging the Python SDK, we can seamlessly integrate with various AWS services such as Lambda, API Gateway, DynamoDB, S3, and others, to develop and deploy efficient and scalable microservices.
  - **Authentication**
    - ∗ **lf1-register:** The register Lambda function is responsible for adding new users to the user pool in AWS Cognito. The function first receives user registration information in the form of an HTTP request. It then processes the information and sends a request to Cognito to create a new user account with the provided information. Once the user is successfully added to the user pool, the function responds with a success message.
    - ∗ **lf1dot5-confirmregistration:** The confirm register Lambda function interacts with AWS Cognito and SES services. Specifically, the function calls the Cognito API to verify the confirmation code sent to the user’s email and activate the user’s account. Then, it uses the SES API to send a confirmation email to the user upon successful registration.
    - ∗ **lf2-login:** The login Lambda function is responsible for verifying the user’s login information using Cognito, and if the credentials are valid, it returns a JWT token that can be used for later user verification.
  - **Profile management**
    - ∗ **lf3-get-user-profile-public:** The get user public profile Lambda function leverages the power of DynamoDB SDK and S3 SDK to fetch user data and profile photo based on the user ID included in the path parameter of the HTTP request.
    - ∗ **lf4-get-user-profile-private:** The get user private profile Lambda function leverages the power of DynamoDB SDK and S3 SDK to retrieve the user data and profile photo using the user ID obtained from the decoded JWT token. This ensures that the private profile can only be accessed by the verified user, as the token contains the necessary information about the user, including the user ID.
    - ∗ **lf5-edit-profile:** The Edit Profile Lambda function utilizes the DynamoDB and S3 SDKs to update the user’s information and profile photo (in base64 format) based on the user ID provided in the decoded JWT token. This ensures that the user editing the profile is verified with their credentials, and only the authorized user can make changes to their profile information.
  - **Matching**
    - ∗ **lf6-produce-match:** The produce match Lambda function calls our algorithm which partitions users into two groups based on matching preference: similar or different. It then encodes similar users into a low dimensional space and runs heuristic clustering using buckets to match within clusters. Any remaining users are randomly matched. Finally, it pushes messages about the matching result to SQS for further processing.
    - ∗ **lf7-past-matches:** The get past match Lambda function retrieves past match results by calling the DynamoDB that contains the match results based on the user ID provided in the decoded JWT

token. The function returns the current or all past matches based on the request made through the HTTP call. It ensures that only the verified user can access their match history using the decoded JWT token that contains the user's credentials.

- ∗ **lf8-submit-review:** The Submit Review Lambda function utilizes the DynamoDB SDK to add the review submitted by the user to the corresponding match based on the match ID from the HTTP request and the user ID from the decoded JWT token. This ensures that the review is associated with the correct match and user.
- ∗ **lf9-pull-send-match-info:** The pull-send-match-info Lambda function is responsible for processing the match results and notifying the users via email. It utilizes the SQS message service to pull messages containing the match results, and then retrieves the corresponding user information from DynamoDB. Using the SES email service, the function sends personalized match result emails to the users based on their information in DynamoDB.

– **Matching Algorithm:** As previously mentioned, users can specify specify features such as "major", "year in school", "interests", etc. In addition, they can specify their matching preference. This field represents if they want to be matched with someone "similar" to them or "different" from them. We assume most people will want to network with people similar to them, though the algorithm accommodates both preferences. We encode users into low dimensional space based on the features they specified in their profile. From there, we run a clustering algorithm using buckets. This works by first generating M buckets based on the range of features seen in the user set. Then it it puts each user in every bucket that they fit into. For example, one bucket might be for users who are strongly technical, such as CS majors, 4th years, etc. We then create pairs of the users with "similar" preference in the same bucket (keeping track to not pair a user up more than once). Then the remaining users are paired such that they avoid those who are in the same bucket. We ran benchmarks on this algorithm on 100,000 users, and the results are displayed in Table I.

- **CloudWatch Schedular:** AWS CloudWatch Scheduler is a service provided by Amazon Web Services (AWS) that allows you to schedule automated actions in the cloud. It allows us to schedule events that trigger automated actions such as starting or stopping an Amazon Elastic Compute Cloud (EC2) instance, or running a Lambda function.

  – **Produce match:** In a real-world deployment scenario, the produce match scheduler is configured to trigger the *lf6-produce-match* Lambda function once a week. This function matches active users based on their preferences using our algorithm. The status of the users is checked before matching to ensure that only active users are matched.

  – **Send match info:** The *lf9-pull-send-match-info* Lambda function is triggered periodically using send match info Scheduler to pull messages containing the match results from the SQS queue containing the match results and send the match result emails to the corresponding users using the SES email service.

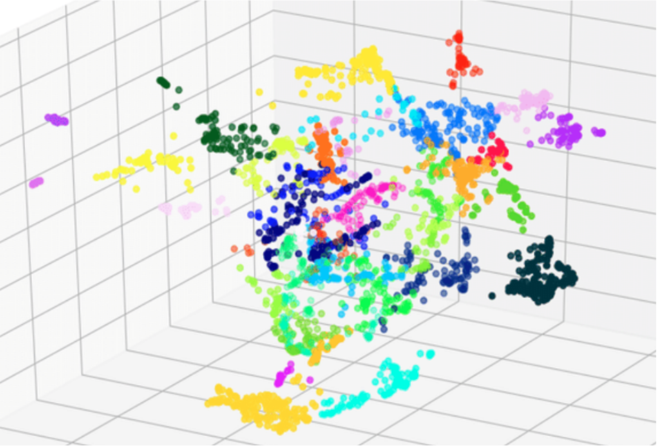


Fig. 2. An overview of our encoded feature of the matching algorithm.

IV. Result

Using our heuristics-based clustering algorithm, we are able to achieve fast matching speeds as shown in Table I due to its linear time complexity in (number of users) * (number of encoding features). Additionally, the algorithm easily adapts to new features that our application may support and gather in the future. A slight downside is that the algorithm, in its current implementation, is unable to accommodate users with very specific preferences. For example, a user might prefer someone in their major but in a different year. However, we made the design decision to sacrifice this bit of flexibility for the speedup as compared to matching algorithms such as Gale–Shapley.

V. Conclusion

In this paper, we detailed the development of our coffee chat platform for Columbia students - Columbia Coffee Chat. We have demonstrated its scalability with increasing users to over 10,000 users, as well as features collected from those users. We show utilization of AWS services to achieve such scalability and ease of development through a loosely coupled microservices-based architecture.

TABLE I
Performance of Matching Algorithm

	"similar" preferences)	**"different" preferences**
100,000 users	∼7.686 seconds	∼5.552 seconds