

CNG 495 – Cloud Computing

Fall 2025

HelpConnect: Cloud-Based Emergency Assistance Platform (Flutter + AWS)

Capstone Project Progress Report

Team Members:

Ece Kocabay **SID: 2591618**

Noyan Saat **SID: 2453504**

Ali Saadettin Yaylagül **SID: 2453637**

Course Code: CNG 495 — **Semester-Year:** Fall 2025 — **Type:** Capstone
Project Progress Report

Date of Submission: November 30, 2025

Contents

1	Topic	3
2	Project Description	3
3	Cloud Delivery Models	4
3.1	Software as a Service (SaaS)	4
3.2	Platform as a Service (PaaS)	4
3.3	Infrastructure as a Service (IaaS)	4
4	Diagrams and Figures	5
4.1	Data Flow Diagram: Context Level (Level 0)	5
4.2	Data Flow Diagram: Level 1	6
4.3	Activity Diagram: Submitting a Help Request	7
5	Computation	8
6	Data Types	9
7	Expected Contribution for Each Member	10
8	Milestones Achieved	11
9	Member Contributions	12
10	Milestones Remaining	14
11	Current UI Screenshots	15
12	Short Tutorials for Cloud Technologies Used	21
12.1	AWS Lambda	21
12.2	Amazon API Gateway	22
12.3	Amazon DynamoDB	23
12.4	Amazon S3	23
12.5	AWS Cognito	24
12.6	Amazon SNS	24
12.7	DynamoDB Streams (Event-Driven Processing)	25

13 GitHub Repository and Source Code Access	25
14 Project Deliverables	26
15 References	28

1 Topic

Client/Server Cloud Application Topic: A cross-platform (mobile/web) emergency assistance and coordination platform developed using a single Flutter codebase for the client and a serverless, event-driven AWS backend.

The platform focuses on **community-driven, non-life-threatening emergencies** across four categories: (1) **medical assistance** such as urgent blood donation or minor injuries, (2) **missing persons and missing pets**, (3) **environmental and neighborhood incidents** including localized flooding or small fires, and (4) **daily life support scenarios** where individuals require quick help from nearby volunteers (e.g., delivering supplies, assisting elderly neighbors, or minor technical/transport issues).

2 Project Description

HelpConnect is a cross-platform emergency assistance application designed to operate seamlessly on both mobile (iOS/Android) and web from a single, unified Flutter codebase. It connects individuals experiencing urgent but manageable emergencies with nearby community volunteers. These individuals are referred to as **Help Seekers**, while those offering assistance are referred to as **Volunteers**.

The client application interacts with the AWS backend through REST APIs exposed by **AWS API Gateway**, which routes requests to serverless **AWS Lambda** functions for business logic execution. Help Seeker and Volunteer data, along with emergency requests, are stored in **Amazon DynamoDB**, while user-uploaded media (e.g., images of the scene, missing pet photos) is saved in **Amazon S3**. User authentication is securely managed by **AWS Cognito**, and real-time push notifications are sent via **Amazon SNS**.

A key feature of the architecture is its event-driven, serverless paradigm. New help requests written to DynamoDB will asynchronously trigger a separate Lambda function via **DynamoDB Streams** to handle the computationally intensive task of matching volunteers. This ensures the Help Seeker receives an immediate response while the heavy processing occurs in the background.

In addition to push notifications, **Volunteers can proactively browse and filter all active emergencies** (e.g., by distance, category, or urgency level) through the application. This design ensures that even if the initially notified volunteer is unavailable or responds late, other

Volunteers can still discover relevant emergencies and offer help. The project also includes a web-based **Admin Dashboard** for monitoring system activity and managing data.

3 Cloud Delivery Models

HelpConnect utilizes a mix of **Software as a Service (SaaS)**, **Platform as a Service (PaaS)**, and **Infrastructure as a Service (IaaS)** layers.

3.1 Software as a Service (SaaS)

Amazon Cognito is used for user authentication and management. It provides a complete identity solution as a managed service, removing the need to build and secure a custom authentication system.

3.2 Platform as a Service (PaaS)

AWS Lambda, **Amazon DynamoDB**, and **Amazon SNS** are the core PaaS components. AWS manages the underlying servers, scaling, and fault tolerance, allowing our team to focus exclusively on the application logic and data models.

3.3 Infrastructure as a Service (IaaS)

Amazon S3 serves as the object storage backbone. While S3 is a managed service, it abstracts the underlying hardware (IaaS), where AWS maintains the physical servers and networking, and our team manages data durability and access configurations.

4 Diagrams and Figures

4.1 Data Flow Diagram: Context Level (Level 0)

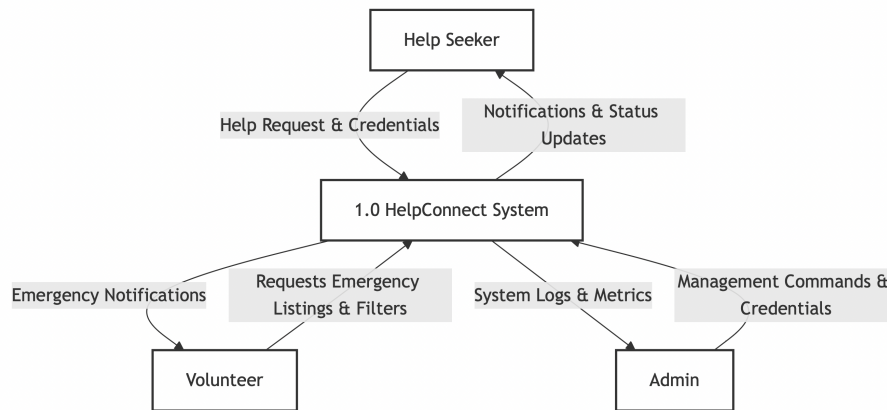


Figure 1: Context Level (Level 0) Data Flow Diagram for the HelpConnect System.

Explanation: This diagram presents the highest-level view of the system. The entire HelpConnect application is shown as a single process (1.0 HelpConnect System). It illustrates the system's boundaries and its primary interactions with external entities: the **Help Seeker**, the **Volunteer**, and the **Admin**. The labeled arrows represent the main categories of data that flow into and out of the system. For example, a Help Seeker provides a Help Request and Credentials and in return receives Notifications and Request Status Updates. Volunteers receive Emergency Notifications and can send Offers to Help or browse open emergencies exposed by the system.

4.2 Data Flow Diagram: Level 1

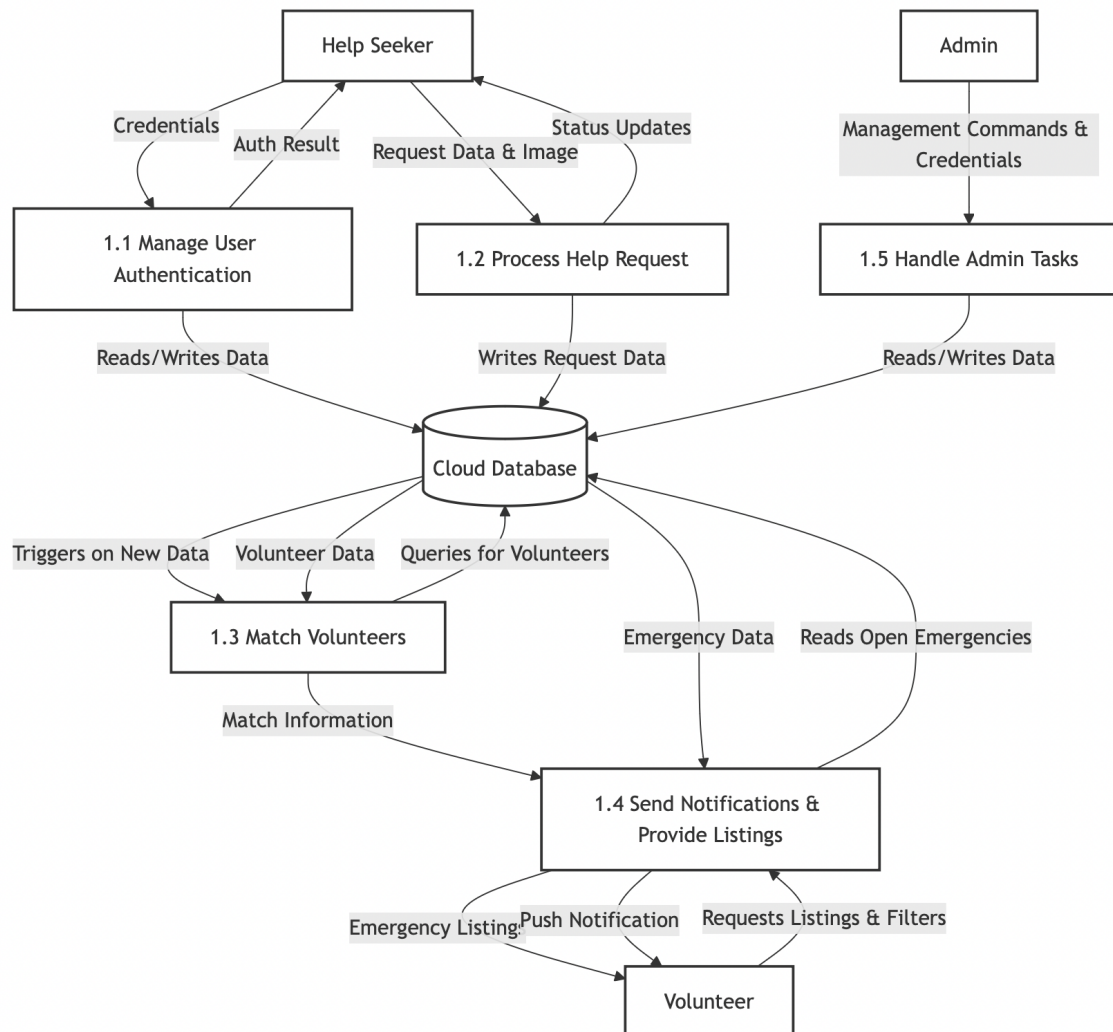


Figure 2: Level 1 Data Flow Diagram showing the major sub-processes.

Explanation: This Level 1 DFD expands the main process into detailed components, showing how data moves between these sub-processes and the central Cloud Database data store. Key processes include Manage User Authentication (1.1), Process Help Request (1.2), Match Volunteers (1.3), and Send Notifications & Provide Listings (1.4). A critical feature illustrated here is the asynchronous flow: the Cloud Database sends a Trigger on New Data flow to the Match Volunteers process, clearly showing the event-driven nature of our architecture, while Volunteers can also request and filter lists of open emergencies independently of the matching process.

4.3 Activity Diagram: Submitting a Help Request

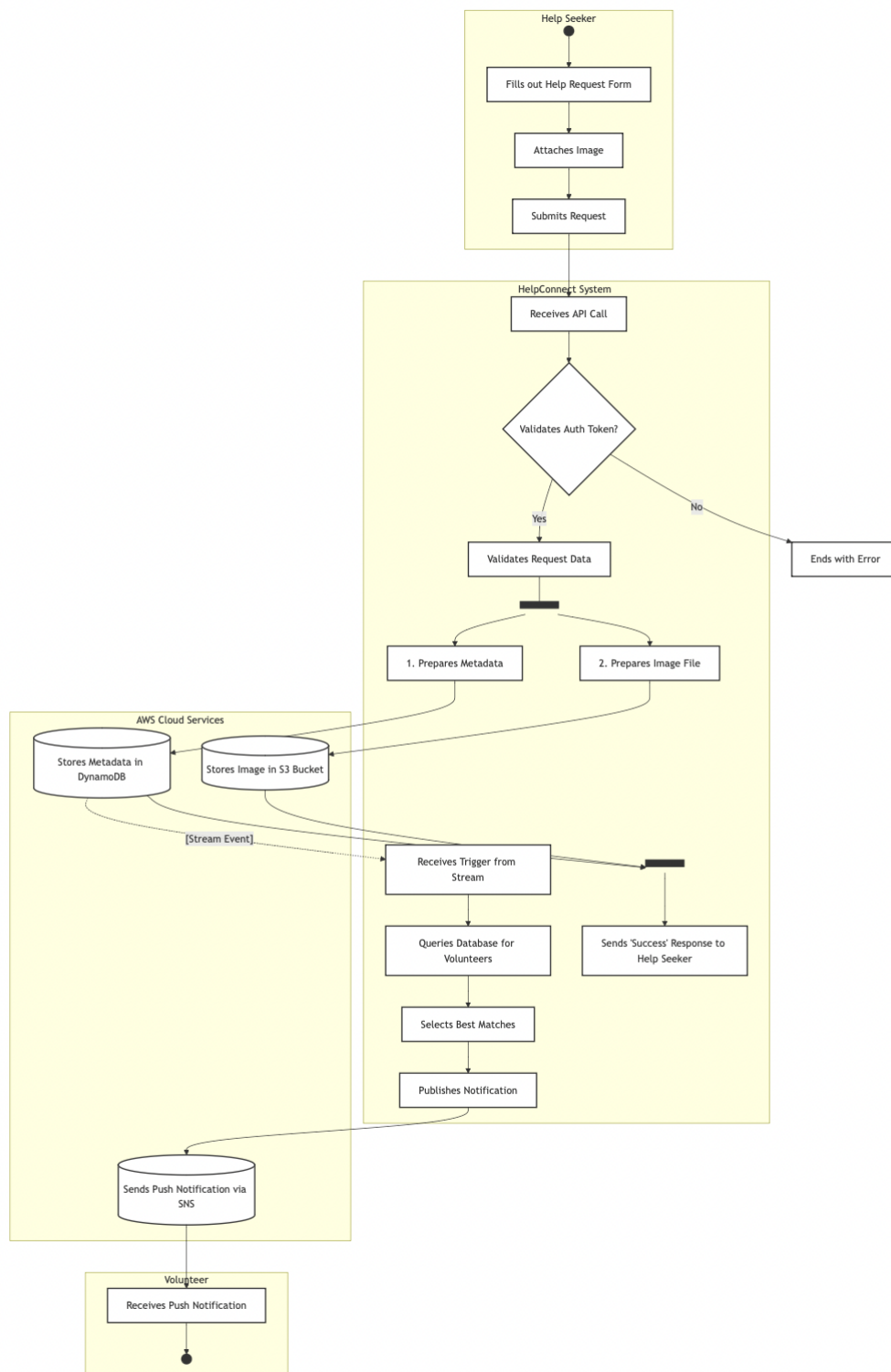


Figure 3: Activity Diagram illustrating the workflow for a Help Seeker submitting a Help Request.

Explanation: This diagram details the step-by-step workflow for the primary use case of submitting a Help Request. The swimlanes clearly define responsibility for each action among the **Help Seeker**, the **HelpConnect System** (our application logic), the underlying **AWS Cloud Services**, and a **Volunteer**. The diagram correctly uses formal UML symbols, including:

- **Decision Node:** To validate the Help Seeker’s authentication token.
- **Fork/Join Bars:** To show that storing metadata and the image file are parallel actions that must both complete before a success response is sent to the Help Seeker.
- **Asynchronous Trigger:** A dotted arrow labeled [Stream Event] shows that the matching logic is triggered in the background by a database event, decoupled from the initial Help Seeker submission. Subsequent Volunteer actions (such as receiving and responding to the push notification) are modeled as separate flows.

5 Computation

The HelpConnect system will perform several key computational tasks to function, primarily within the AWS Lambda functions. These tasks involve processing data to facilitate the matching, listing, and notification processes.

- **Authentication and Authorization:** For every API request, the system will compute the validity of the user’s session by validating the Cognito-issued JWT (JSON Web Token). This ensures that all actions are performed by authenticated and authorized Help Seekers, Volunteers, or Admins.
- **Geospatial Matching Algorithm:** This is the core computation of the application. When a new help request is created, the system will:
 - a. Retrieve the geographic coordinates (latitude and longitude) of the new request.
 - b. Query the DynamoDB database for active Volunteers within a predefined radius.
 - c. Compute a match score for each potential Volunteer based on multiple factors, including distance, urgency level of the request, and matching categories (e.g., medical, environmental, missing pet).
 - d. Rank the results to produce a list of the most suitable candidates to notify.

- **Volunteer Browsing and Filtering:** When a Volunteer opens the emergency list screen, the system will:
 - a. Retrieve all active requests, optionally filtered by geographic radius, category, or urgency.
 - b. Order the results using criteria such as recency or proximity.
 - c. Return a concise representation optimized for mobile devices, enabling Volunteers to proactively select emergencies to respond to, even if they were not part of the initial notification set.
- **Data Validation:** Before any data is written to the database, the backend will perform computational checks to validate the integrity of the incoming data from the client. This includes checking data types, ensuring required fields are present, and sanitizing inputs to prevent errors or security vulnerabilities.
- **Notification Targeting:** After the matching algorithm identifies the best candidates, the system computes the specific notification payload (the message content) and determines the exact list of user device endpoints to send the alert to via Amazon SNS.

6 Data Types

The HelpConnect system will process and store several distinct types of data to function effectively. The primary categories are text, numeric, and binary data, which are handled by different AWS services.

- **Text and Structured Data (JSON):** This is the most common form of data, transmitted as JSON between the Flutter client and the backend API. It includes:
 - **Help Seeker and Volunteer Information:** Cognito user ID, name, email, role (Help Seeker or Volunteer), and user-provided profile details such as skills or preferred help categories.
 - **Request Metadata:** A textual description of the help request, its category (e.g., Medical, Missing Pet, Environmental), urgency level, and current status (e.g., Open, In Progress, Fulfilled).

- **System Metadata:** Unique IDs for requests, timestamps, and log entries.
- **Volunteer Filters and Preferences:** Selected filters (e.g., maximum distance, categories of interest) used when listing emergencies for a Volunteer.
- **Numeric Data:** These values are typically part of the JSON objects but are crucial for computation:
 - **Geographic Coordinates:** Latitude and longitude values used for the geospatial matching algorithm and distance computations.
 - **Timestamps:** Unix timestamps to record when requests are created, updated, or fulfilled.
 - **Scoring Values:** Numerical match scores assigned to Volunteers for a given request.
- **Binary Data:** This refers to non-text files uploaded by users:
 - **Images:** Photos (e.g., JPEG, PNG) uploaded by Help Seekers to provide visual context for their requests (such as a missing dog photo, a flooded street, or fire damage). This binary data is stored directly as objects in Amazon S3.

7 Expected Contribution for Each Member

- **Noyan Saat (Frontend Lead):** Develops the Flutter-based UI for both mobile and web, ensuring an adaptive and responsive user experience. Builds the authentication flow, Help Seeker request forms, Volunteer emergency listing and filtering screens, and the admin dashboard. Integrates the client with the AWS backend via REST APIs.
- **Ece Kocabay (Backend Developer):** Implements the serverless business logic in AWS Lambda. Develops the REST API endpoints and the event-driven matching algorithm triggered by DynamoDB Streams. Designs the DynamoDB data model and configures SNS notification logic, including endpoints for Volunteer browsing and filtering of open emergencies.
- **Ali Saadettin Yaylagül (Cloud Engineer):** Manages the end-to-end AWS cloud infrastructure setup (Cognito, S3, DynamoDB, IAM policies). Configures API Gateway routes and Lambda triggers. Deploys the Flutter Web admin panel via S3 and CloudFront. Ensures the architecture is secure, scalable, and aligned with least-privilege access control.

8 Milestones Achieved

The following timeline summarizes the milestones achieved between 27 October and 30 November 2025, grouped by ODTUClass weeks.

27 Oct – 2 Nov	Finalized project scope and non-life-threatening emergency categories. Designed the overall client–server–cloud architecture. Created the GitHub repository structure for frontend, backend, and documents. Updated the original proposal according to instructor feedback.
3 Nov – 9 Nov	Initialized the Flutter project and basic navigation between main screens. Implemented first versions of Help Seeker and Volunteer home UIs with mock data. Defined core data models (<code>Emergency</code> , <code>Offer</code>) in the Flutter code. Created AWS IAM roles and an S3 bucket to prepare the cloud environment. Created DynamoDB tables <code>HelpRequests</code> and <code>HelpOffers</code> .
10 Nov – 16 Nov	Implemented Lambda functions for creating help requests and listing emergencies. Added a Lambda to list a help seeker’s own requests using a DynamoDB GSI. Configured API Gateway routes for <code>/help-requests</code> , <code>/emergencies</code> and <code>/my-requests</code> . Connected the Flutter frontend to these endpoints via the <code>ApiClient</code> class. Resolved initial IAM and CORS issues between Lambda, DynamoDB, and API Gateway.
17 Nov – 23 Nov	Implemented Lambda functions for creating volunteer offers and listing offers by request. Integrated the “I want to help” flow on the Volunteer side in Flutter. Added a shared Request Detail screen for both Help Seekers and Volunteers. Enabled Help Seekers to see all offers submitted for their requests. Cleaned up placeholder icons/emojis to match the final UI style.
24 Nov – 30 Nov	Completed end-to-end integration of Help Seeker and Volunteer flows with the AWS backend. Fixed DynamoDB Decimal serialization issues by adding JSON conversion logic in Lambdas. Improved loading and error handling in all main Flutter screens. Performed full-system

manual testing across mobile/web UI and cloud services. Prepared the structure of this progress report and collected UI screenshots for documentation.

9 Member Contributions

This section summarizes the individual contributions of each team member during the period 27 October – 30 November 2025. The responsibilities below reflect both the initial role assignments and the concrete tasks completed throughout Stage 2 of the project.

Noyan Saat (Frontend Lead)

- Designed and implemented the Flutter-based user interface for both Help Seeker and Volunteer roles.
- Developed all major screens: Role Selection, Help Seeker Home, Volunteer Home, My Requests, Create Help Request, and the shared Request Detail page.
- Built request creation workflows including form validation, category/urgency selection, and submission to AWS backend.
- Integrated all frontend components with the serverless backend using the `ApiClient` class.
- Implemented filtering logic on the Volunteer side (category, urgency, dynamic list updates).
- Converted all mock-data UI elements to live data driven by API Gateway.
- Removed temporary icons/emojis and redesigned UI components to match project visual standards.
- Conducted full manual testing on iOS simulator and prepared screenshots for documentation.

Ece Kocabay (Backend Developer)

- Designed and implemented the serverless backend logic using AWS Lambda and Python.
- Developed all REST API endpoints required for Stage 2:
 - POST /help-requests — create new help requests.
 - GET /emergencies — list all active emergencies.
 - GET /my-requests — list help seeker's own requests via GSI.
 - POST /offers — create volunteer offers.
 - GET /offers?requestId=... — list volunteer offers for a request.
- Implemented DynamoDB data conversion, validation, and error-handling logic (e.g., Decimal-to-int conversion).
- Designed the DynamoDB schema for both HelpRequests and HelpOffers tables.
- Implemented Lambda response formatting and CORS-compliant response headers.
- Debugged and resolved backend issues including IAM permission errors, route misconfiguration, and request parsing bugs.
- Performed integration testing with the Flutter frontend to ensure end-to-end correctness.

Ali Saadettin Yaylagül (Cloud Engineer)

- Set up the AWS environment including IAM roles, Lambda execution policies, and access roles for DynamoDB/S3/API Gateway.
- Created and configured DynamoDB tables with primary keys and indexes (help_seeker_id-index, request_id partition key).
- Configured API Gateway HTTP API, attached routes to Lambda functions, and resolved CORS settings.
- Set up the S3 bucket for project storage and future deployment needs.
- Managed Lambda–DynamoDB–API Gateway integration permissions and execution roles.

- Ensured all cloud components adhered to least-privilege IAM principles and followed architectural guidelines set in the project proposal.
- Coordinated deployment testing of the full cloud workflow, verifying logs, metrics, and end-to-end stability.

10 Milestones Remaining

1 Dec – 7 Dec	Implement AWS Cognito user pool and app client for authentication. Integrate login and registration flows in the Flutter frontend with Cognito (Noyan). Add role-aware navigation (Help Seeker / Volunteer / Admin) based on Cognito user attributes (Noyan). Secure existing API Gateway endpoints with Cognito authorizers and update Lambda functions to validate JWT tokens (Ece). Refine IAM roles and permissions to follow least-privilege access for authenticated users (Ali).
8 Dec – 14 Dec	Design and implement the initial volunteer matching Lambda triggered by DynamoDB Streams (Ece). Configure DynamoDB Streams on the <code>HelpRequests</code> table and connect them to the matching Lambda (Ali). Integrate Amazon SNS to send notifications to selected volunteers when a new help request is created (Ali). Extend the Flutter client to display basic notification-related UI hooks (e.g., placeholders for incoming alerts) (Noyan). Begin drafting final system architecture diagrams and update documentation according to the new components (Ali).
15 Dec – 23 Dec	Add an Admin Dashboard view in Flutter Web for monitoring requests, offers, and basic metrics (Noyan). Finalize and tune the matching logic (e.g., distance, urgency, category) and log matching decisions for debugging (Ece). Prepare deployment scripts / configuration for a demo-ready environment (Lambda, API Gateway, DynamoDB, S3, Cognito, SNS) (Ali). Conduct end-to-end testing of all user flows (registration, login, help request creation, volunteer matching, offers, and admin monitoring) (Ali). Finalize the project README, clean

up the repository structure, and prepare the final report and demo presentation materials (All).

11 Current UI Screenshots

This section presents the current state of the Flutter user interface as implemented in Stage 2. Screenshots include authentication pages, Help Seeker flows, Volunteer flows, and shared request detail views. All screenshots were captured from the iOS simulator.

Authentication Screens

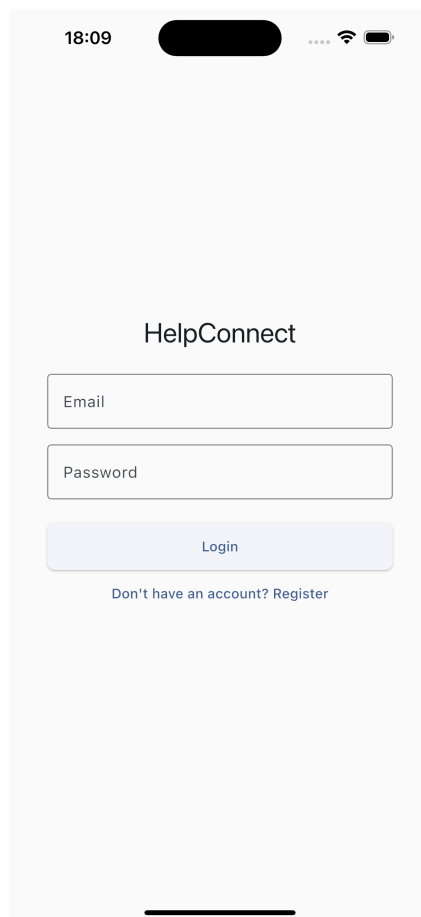
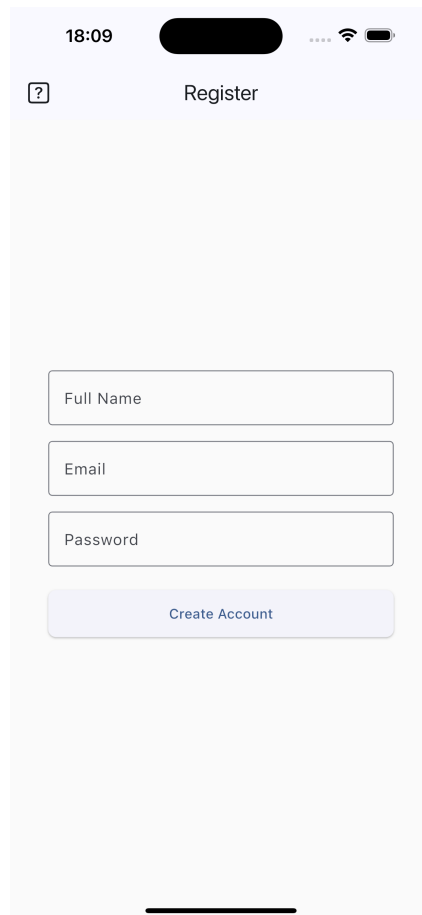


Figure 4: Login Screen allowing registered users to authenticate.



The image shows a mobile application registration screen. At the top, the status bar displays the time 18:09, a black pill-shaped notch, and icons for cellular signal, Wi-Fi, and battery. Below the status bar, there is a header area with a question mark icon in a square on the left and the word "Register" in the center. The main content area contains three stacked text input fields labeled "Full Name", "Email", and "Password". Below these fields is a light blue button with the text "Create Account". The entire screen has a light gray background.

Figure 5: Registration Screen for creating new user accounts.

Help Seeker Screens

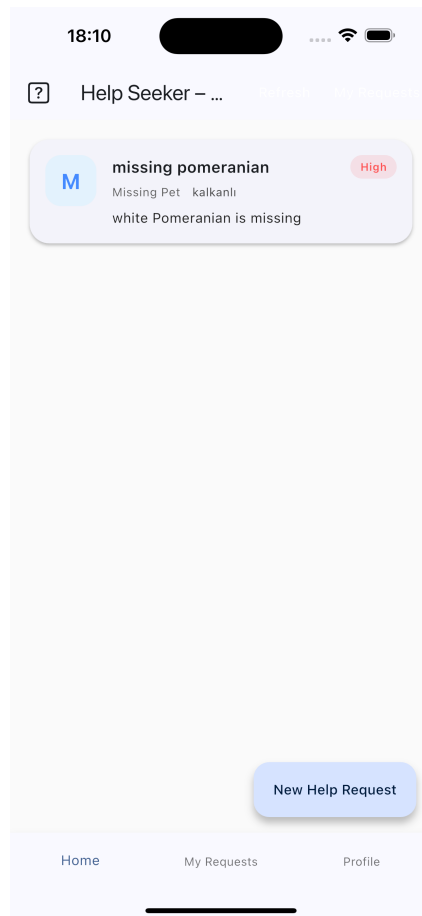


Figure 6: Help Seeker Home Screen displaying active emergencies.

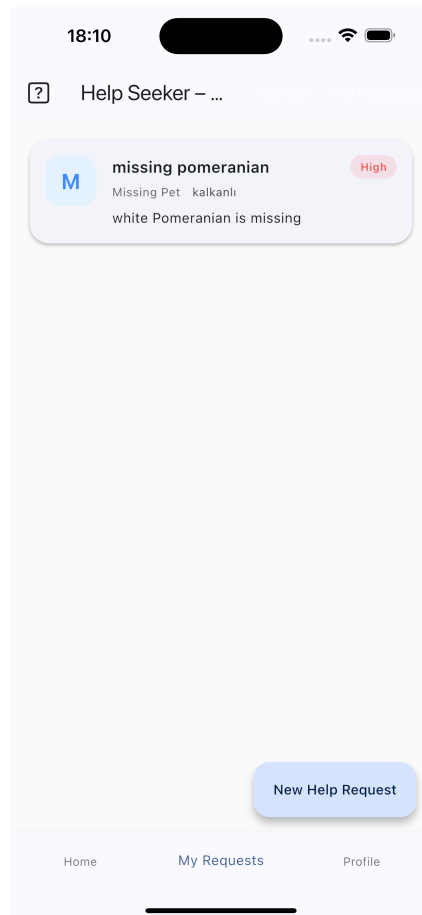


Figure 7: My Requests Screen showing the Help Seeker's submitted requests.

The image shows a mobile application interface for creating a new help request. At the top, the status bar displays the time 18:10, a black pill-shaped notch, and icons for cellular signal, Wi-Fi, and battery. Below the status bar is a header with a question mark icon and the text 'New Help Request'. The form consists of several input fields: a 'Title' field, a larger 'Description' field, a 'Location' field, a 'Category' dropdown menu with 'Medical' selected, and an 'Urgency' dropdown menu with 'High' selected. Each dropdown menu has a question mark icon on the right. At the bottom of the form is a light blue button labeled 'Submit Help Request'. The background of the app is a light gray, and there is a black horizontal bar at the very bottom of the screen.

18:10

New Help Request

Title

Description

Location

Category

Medical

Urgency

High

Submit Help Request

Figure 8: Create Help Request Form with category, urgency, and description fields.

Volunteer Screens

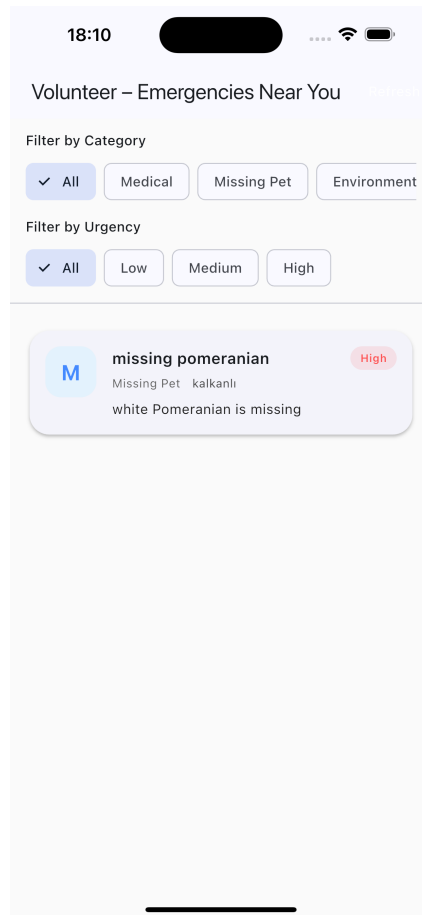


Figure 9: Volunteer Home Screen with emergency listings.

Shared Screens

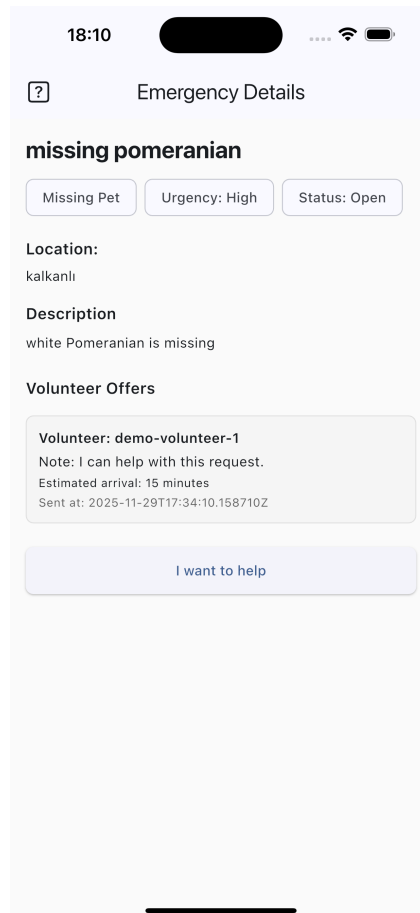


Figure 10: Request Detail Screen showing full information about a help request only difference as volunteer they can see a button that says I want to help which is not visible to helpseekers.

12 Short Tutorials for Cloud Technologies Used

This chapter provides short, practical tutorials for the main AWS cloud technologies used in the HelpConnect platform. Each subsection covers what the service is, why we used it, and a minimal example showing how it integrates with our project.

12.1 AWS Lambda

What it is: AWS Lambda is a serverless compute service that allows running code without provisioning or managing servers. It executes functions in response to events or direct API calls.

Why we used it: All backend logic (creating requests, listing emergencies, submitting offers) is written as Python-based Lambda functions. Lambda also processes background

events triggered by DynamoDB Streams for asynchronous volunteer matching.

How to create a Lambda:

1. Open AWS Console → Lambda → Create function.
2. Choose Author from scratch, runtime = Python 3.12.
3. Attach an execution role that allows DynamoDB access.
4. Paste Lambda code and deploy.

Example (simplified):

```
import json
def lambda_handler(event, context):
    return {
        "statusCode": 200,
        "body": json.dumps({"message": "Hello from Lambda!"})
    }
```

12.2 Amazon API Gateway

What it is: API Gateway exposes HTTPS endpoints that trigger Lambda functions.

Why we used it: Every operation in HelpConnect (create request, list emergencies, offer help) is accessed through REST API routes defined in API Gateway.

How to create an API route:

1. Navigate to API Gateway → HTTP API.
2. Add route, e.g., POST /help-requests.
3. Attach the target Lambda function.
4. Enable CORS for Flutter mobile/web.

Example Route Mapping:

```
POST /help-requests  --->  Lambda: CreateHelpRequest
GET  /emergencies   --->  Lambda: ListEmergencies
POST /offers         --->  Lambda: SendOffer
```

12.3 Amazon DynamoDB

What it is: A fully managed NoSQL database offering single-digit millisecond latency.

Why we used it: We store Help Requests, Volunteer Offers, and (later) User Profiles. DynamoDB Streams notify background Lambdas when new requests are added.

Key tables:

- HelpRequests (PK: request_id)
- HelpOffers (PK: offer_id, GSI: request_id)

Minimal put-item example:

```
table.put_item(Item={
    "request_id": "123",
    "title": "Urgent Help Needed",
    "urgency": "High"
})
```

Minimal query example (GSI):

```
table.query(
    IndexName="request_id-index",
    KeyConditionExpression=Key("request_id").eq("123")
)
```

12.4 Amazon S3

What it is: A scalable object storage service used for uploading images, documents, or web assets.

Why we used it: S3 stores help request images (e.g., missing pet photos) and will host the admin dashboard deployment.

Basic bucket creation:

1. S3 → Create bucket
2. Disable public access only if using CloudFront
3. Enable CORS for Flutter uploads

Python upload example:

```
s3 = boto3.client("s3")
s3.upload_file("dog.jpg", "helpconnect-bucket", "images/dog.jpg")
```

12.5 AWS Cognito

What it is: Managed authentication service with sign-up, login, and identity tokens (JWT).

Why we will use it (Stage 3): Cognito will control:

- User registration and login
- Distinguishing Help Seekers, Volunteers, Admins
- Generating ID tokens used by Flutter to authenticate API calls

User Pool creation steps:

1. Cognito → Create user pool
2. Enable email-based registration
3. Configure App Client (Flutter)
4. Enable hosted UI if desired

12.6 Amazon SNS

What it is: Notification service for sending mobile push or SMS messages.

Why we will use it (Stage 3): SNS will notify nearby volunteers immediately when a Help Seeker submits a new request.

Message publish example:

```
import boto3
sns = boto3.client("sns")
sns.publish(
    TargetArn=endpoint_arn,
    Message="New emergency nearby!"
)
```

12.7 DynamoDB Streams (Event-Driven Processing)

What it is: A change-log stream that triggers Lambda functions whenever new data is added.

Why we will use it (Stage 3): When a Help Request is created:

1. DynamoDB Streams detects the new item.
2. Triggers the MatchVolunteersLambda.
3. Lambda runs the matching algorithm.
4. SNS sends notifications automatically.

Enabling Streams:

1. DynamoDB → Table → Exports and streams.
2. Enable Stream: NEW_IMAGE.
3. Attach a Lambda trigger.

13 GitHub Repository and Source Code Access

All source code developed for the HelpConnect project is publicly available on GitHub. The repository contains the Flutter client application, AWS Lambda backend functions, API Gateway configurations, DynamoDB schema information, and all project documentation (proposal, progress report, and final report).

GitHub Repository URL: <https://github.com/ecekocabay/helpconnect.git>

Clone the repository using:

```
git clone https://github.com/ecekocabay/helpconnect.git
```

Repository Structure

The repository is organized into clear directories to separate client, backend, cloud infrastructure, and documentation:

- /frontend/ – Flutter mobile/web application.
- /backend/ – AWS Lambda Python functions and API routes.

- `/documents/` – Project proposal, progress report, final report, diagrams.
- `README.md` – High-level project summary and setup instructions.

This structure ensures reproducibility and clarity for both instructors and future developers. All code referenced in this report points to functions, classes, and modules maintained in this repository.

14 Project Deliverables

This section summarizes the main deliverables that will be included with the completed Help-Connect system. All materials will be uploaded to GitHub and referenced in the final report and README.

1. Client Deliverables (Flutter App)

- Full Flutter codebase for mobile and web.
- Help Seeker features: Create Request, My Requests, Request Detail.
- Volunteer features: Emergency listing, filtering, and offering help.
- Authentication UI (Login and Registration).
- Admin dashboard (web).

2. Backend Deliverables (AWS Lambda + API Gateway)

- Lambda functions for:
 - Creating requests
 - Listing emergencies
 - Listing user requests
 - Creating offers
 - Listing offers by request
 - (Stage 3) Matching volunteers via DynamoDB Streams
- API Gateway with all REST endpoints and CORS configuration.

3. Database Deliverables (DynamoDB)

- HelpRequests table with GSI for help seeker ID.
- HelpOffers table with GSI for request ID.
- Exported DynamoDB backup (JSON) for demo.

4. Storage Deliverables (Amazon S3)

- S3 bucket for help request images.
- S3 bucket (or folder) for hosting the Flutter Web admin panel.

5. Cloud Infrastructure Deliverables

- IAM roles and execution policies (least privilege).
- Cognito User Pool configuration (Stage 3).
- SNS setup for volunteer notifications (Stage 3).
- CloudWatch logs for Lambda monitoring.

6. Documentation Deliverables

- Project proposal, progress report, and final report.
- UML diagrams and system architecture figures.
- Setup guide for both frontend and backend.
- GitHub repository with README and source code.

7. Demo Deliverables

- Screenshots and video demonstration.
- Test data and DynamoDB exports used during the demo.

15 References

- Amazon Web Services (AWS) Documentation – Lambda, DynamoDB, S3, Cognito, SNS, CloudFront: <https://docs.aws.amazon.com>
- Flutter Official Documentation: <https://docs.flutter.dev>
- Unified Modeling Language (UML) Specification: <https://www.omg.org/spec/UML/>
- CNG 495 Capstone Project Instructions (Fall 2025–2026)