Wagwan.london: Detailed Instructions

1. Prerequisites

- **AWS Account**: You need an AWS account with sufficient permissions (IAM roles/policies) to create and configure S3, CloudFront, Lex, Lambda, and optionally API Gateway or Bedrock.
- **Basic Familiarity with the AWS Console**: Knowing how to navigate the console is helpful, but we'll walk through the specifics.
- **Local Files**: You already have your HTML, CSS, and images ready, or you can create them based on the examples provided below.

Screenshot Note: Throughout this guide, you'll see "Take a screenshot" prompts. Use these when you want to document each step for future reference. Feel free to rename them or follow any numbering convention you prefer.

2. Overview of the Architecture

1. **Static Front End**: A single-page HTML/CSS chatbot interface hosted on S3 and delivered via CloudFront.
2. **Chatbot Logic**:
   - **Amazon Lex** handles natural language understanding.
   - **AWS Lambda** is triggered by Lex to call **Bedrock** (Anthropic Claude) for deeper conversational abilities.
3. **Data Flow**:
   - User opens your webpage (served by CloudFront).
   - Front-end JavaScript sends user messages to Lex (via API Gateway or Lex runtime endpoints).
   - Lex calls Lambda, which calls Bedrock, then returns the response back through Lex → front end.

3. Setting Up AWS Services

3.1 Create an S3 Bucket (Static Website)

1. **Go to the S3 Console**
   - URL: https://s3.console.aws.amazon.com/s3
2. **Click "Create bucket."**
   - **Bucket name**: For example, `wagwanlondon-frontend-bucket`.
   - **Region**: Choose one close to you or your users.

    - Leave other settings at defaults or configure as needed.
3. **Click "Create bucket."**

Take a screenshot of your newly created bucket. Example name: `01-create-s3-bucket.png`

3.2 Configure Permissions & Static Website Hosting

1. **Open the bucket** you just created.
2. **Enable Public Access** or set up a **Bucket Policy** (depends on how you want to control access). For a public static site, you typically:
    - Go to **Permissions → Public access settings**.
    - Uncheck "Block all public access."
    - Confirm.
3. **Static Website Hosting**
    - Go to **Properties** tab.
    - Scroll down to **Static website hosting**.
    - Enable it, set the **Index document** to `index.html`.
    - Click **Save**.

Take a screenshot of your static website hosting configuration. Example name: `02-static-website-hosting.png`

3.3 (Optional) Request a Custom Domain & SSL Certificate

If you plan to use a custom domain like `wagwanlondon.com`:

1. **Obtain/Retrieve the Domain** (through Route 53 or another domain registrar).
2. **AWS Certificate Manager (ACM)**
    - Go to https://console.aws.amazon.com/acm/home.
    - Request a public certificate for your domain (e.g., `*.wagwanlondon.com` or `wagwanlondon.com`).
    - Validate via DNS or Email. DNS is recommended.
3. Keep track of the **ARN** of your certificate for the next step.

Take a screenshot of the certificate request. Example name: `03-acm-certificate.png`

3.4 Create a CloudFront Distribution

1. **Go to CloudFront**: https://console.aws.amazon.com/cloudfront/v3/home.
2. **Click "Create Distribution."**

3. **Origin domain**: Choose your S3 bucket. The S3 domain typically appears in the dropdown.
4. **Default Cache Behavior**:
    - Keep defaults for now (you can optimize later).
5. **SSL Certificate**:
    - Under “Settings,” choose **Custom SSL Certificate** and select the one from ACM (if you set one up).
6. **Alternate Domain Names (CNAMEs)**:
    - Enter your domain if using a custom domain (otherwise skip).
7. **Create Distribution**.

Take a screenshot of the CloudFront distribution overview. Example name: `04-cloudfront-settings.png`

CloudFront will now create and deploy. This can take ~15–30 minutes. After that, you’ll see a **Domain Name** in the CloudFront console. You can use that to access your site.

4. Building Your Chatbot Web Page

Since we’re skipping React and going with a simpler HTML/CSS setup, you can store all your files in a small local folder structure like this:

```
wagwanlondon-frontend/
|-- index.html
|-- style.css
|-- images/
|     |-- wagwan_background.jpg
|     |-- wagwan_mobile_background.jpg
```

4.1 Project Folder Structure

- **index.html**: Main chatbot page.
- **style.css**: Styles for layout and chat display.
- **images/**: Contains your desktop and mobile background images (or any other images you need).

4.2 HTML File (index.html)

Below is the code you provided, with minimal modifications (like ensuring your script block is placed at the end):

html

```
<!DOCTYPE html>
<html>
<head>
  <title>Chatbot Hoodie</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div id="hoodie-container">
    <div id="chatbot-window">
      <div id="chat-messages-container"></div>
      <input type="text" id="user-input" placeholder="Type
your message...">
      <button id="send-button">Send</button>
    </div>
  </div>

  <script>
    function addNewMessage(message, isUser) {
      const chatWindow = document.getElementById('chat-
messages-container');
      const messageDiv = document.createElement('div');
      messageDiv.classList.add('message');
      messageDiv.classList.add(isUser ? 'user' : 'bot');
      messageDiv.textContent = message;
      chatWindow.appendChild(messageDiv);
      chatWindow.scrollTop = chatWindow.scrollHeight;
    }

    // Initial bot greeting
    addNewMessage("Hello! How can I help you?", false);

    const sendButton = document.getElementById('send-
button');
    const userInput = document.getElementById('user-
input');

    // Send button functionality
    sendButton.addEventListener('click', () => {
      const message = userInput.value;
      if (message.trim() !== "") {
        addNewMessage(message, true);
        userInput.value = "";

        // Here is where you would typically call your API
        // For now we simulate a response:
        setTimeout(() => {
          const botReply = "I received your message: " +
message;
          addNewMessage(botReply, false);
        }, 500);
      }
    });

    // Press Enter key to send
    userInput.addEventListener("keyup", function(event) {
      if (event.keyCode === 13) {
        sendButton.click();
      }
    });
  </script>
</body>
</html>
```

4.3 CSS File (style.css)

```
css

body {
  margin: 0;
  overflow: hidden;
  font-family: sans-serif;
}

#hoodie-container {
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
  background-image: url("images/
wagwan_mobile_background.jpg");
  background-size: cover;
  background-position: center;
  background-repeat: no-repeat;
  position: relative;
}

#chatbot-window {
  position: absolute;
  background-color: black;
  color: white;
  padding: 20px;
  border-radius: 10px;
  box-sizing: border-box;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  width: 70%;
  max-width: 400px;
  height: 37.5%;
  overflow-y: auto;
  max-height: 100%;
}

@media (min-width: 768px) {
  #hoodie-container {
    background-image: url("images/wagwan_background.jpg");
  }

  #chatbot-window {
    width: 37.5%;
    max-width: 375px;
    height: 37.5%;
    padding: 5px;
    overflow-y: auto;
    max-height: 100%;
  }
}
```

```
@media (min-width: 1920px) {
  #chatbot-window {
    max-width: 700px;
  }
}

.message {
  margin: 10px 0;
  padding: 10px;
  border-radius: 5px;
}
.message.user {
  background-color: #444;
  text-align: right;
}
.message.bot {
  background-color: #333;
  text-align: left;
}
```

4.4 Images

Place your **wagwan_mobile_background.jpg** and **wagwan_background.jpg** in the `images/` folder. Adjust references in `style.css` if the folder is named differently.

5. Uploading and Testing Your Static Site on S3

1. **Open your bucket** in the S3 console.
2. Click **Upload → Add files** (select `index.html`, `style.css`), and **Add folder** if you want to upload `images/`.
3. After uploading, **make the files publicly readable** (if you have not set the entire bucket to public). You can do this by adjusting the bucket policy or individual object permissions.
4. **Test** by accessing the **S3 Website Endpoint** (shown under Static Website Hosting in the bucket's properties). If everything is correct, your simple chatbot page should appear.

Take a screenshot of the successful S3 upload or the website in your browser. Example name: `05-s3-website-success.png`

5.1 CloudFront Testing

- If you're using CloudFront, wait for the distribution to deploy.
- Then open the **CloudFront domain** (e.g., `d12345abcdef.cloudfront.net`) in your browser.
- You should see the chatbot page.
- If you set up a custom domain with Route 53, confirm your DNS records point to the CloudFront distribution. Then access via `https://your-custom-domain.com`.

Take a screenshot of the site loading from CloudFront. Example name: `06-cloudfront-site.png`

6. Amazon Lex and AWS Lambda Setup

Now for the chatbot logic. You want Lex to interpret user messages, call Lambda, which uses Bedrock (Anthropic Claude), and return a response.

6.1 Create an Amazon Lex Bot

1. **Go to Amazon Lex**: https://console.aws.amazon.com/lexv2
2. **Create a bot**
   - Specify a name (e.g., `WagwanlondonBot`).
   - For "Output voice," you can choose none if you're just using text.
   - Set the language(s) you need.
3. **Intents**:
   - Create a simple intent, or let the console create a sample one for you.
   - This is where you specify sample utterances.
4. **Save & Build** the bot.
5. **Test** in the Lex console to ensure it responds with a basic greeting or default message.

Take a screenshot of the Lex bot creation page. Example name: `07-lex-bot-creation.png`

6.2 Create an AWS Lambda Function

1. **Go to Lambda**: https://console.aws.amazon.com/lambda
2. **Create function**
   - Name it (e.g., `WagwanlondonFulfillment`).
   - Runtime: Python 3.9 (or whichever is supported).
   - Permissions: Make sure it can call Bedrock (or you can add this policy later).
3. **In the Function code** (inline editor or your favorite dev environment), add logic:
   - Parse the Lex event.

- Call **Amazon Bedrock** with Anthropic Claude.
- Return a structured response that Lex can parse.

A simplistic Python snippet might look like:

```
python

import json
import boto3

def lambda_handler(event, context):
    user_input = event["inputTranscript"]

    # Example: call Bedrock client (pseudo-code)
    # bedrock = boto3.client('bedrock', region_name='us-
east-1')
    # response = bedrock.invoke_model(
    #     modelId="anthropic.claude-v1",
    #     content=user_input
    # )
    # raw_answer = response["body"]  # or wherever the data
is

    raw_answer = f"Your input was: {user_input}. (Claude's
response goes here)"

    # Format a response for Lex
    return {
        "sessionState": {
            "dialogAction": {
                "type": "Close"
            },
            "intent": {
                "name": "YourIntentName"
            }
        },
        "messages": [
            {
                "contentType": "PlainText",
                "content": raw_answer
            }
        ]
    }
```

Take a screenshot of your Lambda function's code or settings. Example name: `08-lambda-code.png`

6.3 Integrate Lex with Lambda

1. **Return to Lex** → Your bot → **Intents**.
2. Under **Fulfillment** (or "Lambda function"), specify the Lambda you created.
3. Build/Deploy your bot again.
4. **Test** in the Lex console. If everything is correct, you should see your Lambda's response.

Take a screenshot of the Lex → Lambda fulfillment setting. Example name: `09-lex-lambda-integration.png`

7. Connecting Your Front End to Lex (via API Gateway)

Your front end's JavaScript can call Lex directly (through Lex runtime APIs) or via an **API Gateway** that triggers Lex. Many prefer an API Gateway → Lambda → Lex approach, but you can also use Lex's built-in web socket interface. For a straightforward solution:

1. **Create an API Gateway** (REST or HTTP API)
    - Under "Integrations," choose your Lambda that calls Lex. Alternatively, you can have the Lambda logic call `RuntimeV2` in Lex.
2. **Deploy** your API Gateway to a stage, e.g. `/prod`.
3. **Note the endpoint**: Something like `https://xyz123.execute-api.us-east-1.amazonaws.com/prod`.
4. **In your index.html** `setTimeout` block, replace the dummy code with a real fetch:`js`

```
fetch("https://xyz123.execute-api.us-east-1.amazonaws.com/prod", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ userInput: message })
})
  .then(response => response.json())
  .then(data => {
    // data should have the bot's message
    addNewMessage(data.botReply, false);
  })
  .catch(err => console.error(err));

```

Take a screenshot of the API Gateway integration or your final fetch code. Example name: `10-api-gateway-setup.png`

8. Validation & Testing

1. **Open your CloudFront URL** (or S3 website link) in the browser.
2. **Type a message** in the chat input.
3. **Watch your Lambda logs** (CloudWatch) to confirm the request flows:
   - `index.html` → fetch call → API Gateway → Lambda → Lex → (optionally Bedrock) → response.
4. **Observe the chatbot** responds with the message from your Lambda/Claude.

Take a screenshot of the working conversation. Example name: `11-chatbot-conversation.png`

9. Screenshot Reference

If you're documenting the entire process, you might have screenshots named:

1. `01-create-s3-bucket.png`
2. `02-static-website-hosting.png`
3. `03-acm-certificate.png`
4. `04-cloudfront-settings.png`
5. `05-s3-website-success.png`
6. `06-cloudfront-site.png`
7. `07-lex-bot-creation.png`
8. `08-lambda-code.png`
9. `09-lex-lambda-integration.png`
10. `10-api-gateway-setup.png`
11. `11-chatbot-conversation.png`

10. Next Steps

1. **Refine Your Chatbot**

    - Add more Lex intents, better fallback handling, advanced logic in Lambda, or deeper context with Bedrock.
2. **Add Cognito Authentication**
    - If you want to restrict usage to certain users, integrate Amazon Cognito for secure access.
3. **Improve UI**
    - Show typing indicators, animations, or richer media responses.
4. **Logging & Analytics**
    - Track conversation metrics in CloudWatch or store them in DynamoDB to analyze usage.
5. **Continuous Deployment**
    - Automate your build and deployment with a CI/CD pipeline so updates go live without manual steps.

Conclusion

You've created a robust one-page chatbot site, hosted on AWS S3 and served by CloudFront, integrated with Amazon Lex, AWS Lambda, and Bedrock/Anthropic Claude. The detailed steps here, combined with the suggested screenshots, should help you produce thorough documentation suitable for someone with minimal AWS experience.

When you're done, you'll have a fully functional and documented chatbot project that can be easily shared, tested, and scaled.