Open questions:

1. Do we want to train the model inside or outside AWS?
2. Are we prioritizing ease of deployment over cost for MVP, given that we have less than 10 weeks to create this?
3. Do we want to use an OpenAI model or instead use a model that we could host ourselves, such as Llama 2? If we could host this LLM, e.g. a HuggingFace model, as a Sagemaker model in our AWS account, this will probably reduce latency and eliminate out-of-pocket costs for our team.

TODOs:

1. Update the sequence diagram with the UI/API layers, reached out to Isabel to get the original that I can edit.
2. Update option 3 of design if we don't want to go with option 1.
3. I think we should use one account for everything and have different logins set up with roles. Once we get the credits into one account I can set this up.
4. I need the model inputs and outputs and an example conversation in the UI with how we want to handle mistakes to be able to fully form the API contract (i.e. need to know what information is needed from the front-end for context).
5. Find out if we can use Figma for free.

Project LangBot: High Level Design

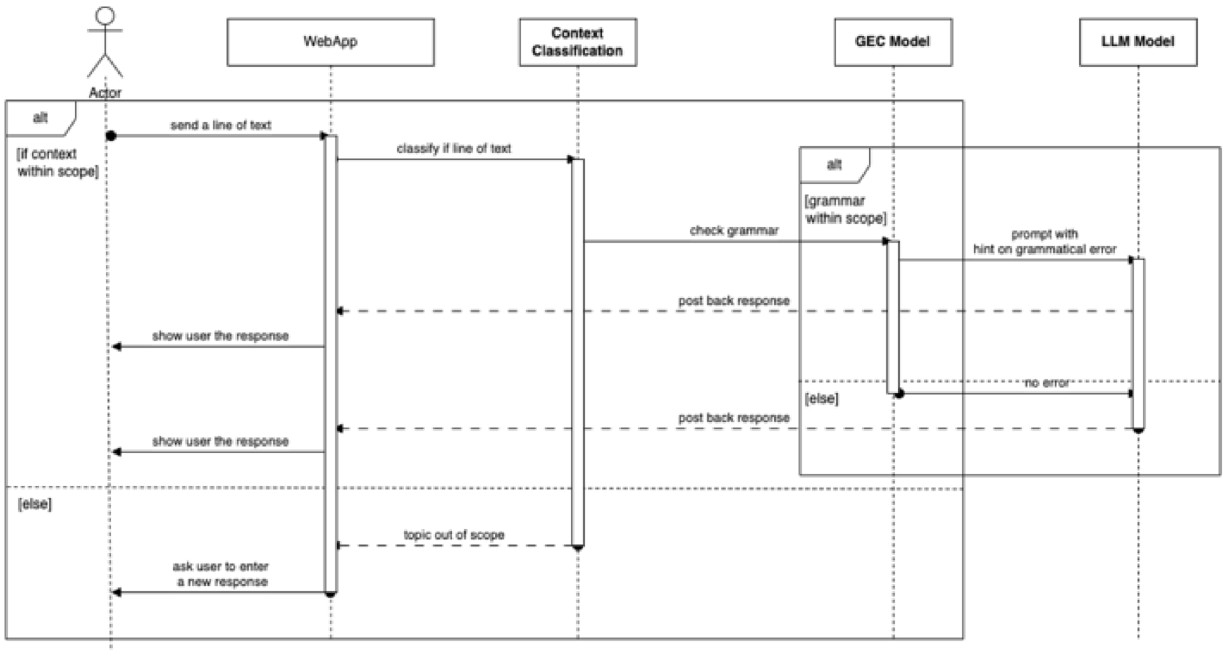
Requirements

This document is designing for the Project Langbot MVP requirements, defined at Project Langbot: Product Roadmap, while keeping the P1 requirements in mind as well. The MVP requirements are as follows at time of writing:

- Project Langbot will offer beginner lessons in Spanish in web app form. We would like to offer at least 2 lessons.
- Langbot will offer cue based feedback to users, instead of just correcting them directly.
- If a user answers a question incorrectly 3 times, Langbot will give them the right answer and move on. "Incorrect" means that the answer is not within context or the grammar is incorrect.
- Once a lesson is completed, the UI will signify that the lesson is complete using a celebratory modal or other UI indicator.
- The Langbot web app will include a home screen that describes the product and team; a "lobby" where users can view the lessons and choose the lesson that they want to start; a screen that will demo the example conversation once they pick a lesson; and the screen for chatting.
- There will be no login or user personalization for MVP.
- Latency is important. We want to design for low latency chatbot interactions.

Components

Sequence Diagram



TODO update this diagram to include the UI/API components

Infrastructure

We will be using AWS for our model and software infrastructure. There are a few different design options that we could use for our system architecture, depending on where we want to do model training and whether we want to prioritize ease of developer experience or cost savings. I will detail these options in the Architecture Options section, but will first detail a framework that could be useful for us.

AWS Amplify

I would like to leverage AWS Amplify to generate and manage our software components and continuous deployment pipeline.

AWS Amplify is a framework offered by AWS to develop web applications. From AWS: “Amplify is a framework with a set of tools and services that help you build **scalable full stack applications**, powered by AWS services. AWS Amplify is a set of tools and services that can be used together or on their own, to help front-end **web and mobile** developers build scalable full stack applications, powered by AWS. With Amplify, you can configure app backends and connect your app in **minutes**, deploy **static web apps** in a few clicks, and easily manage app content outside the AWS console."

AWS Amplify is an abstraction for setting up a web app and its deployment pipelines quickly. It will also set up continuous deployment for us so that when we push to a software component repo, it will deploy as well. We will still need a model deployment pipeline that will be separate from the application deployment pipeline.

AWS Amplify supports creating iOS, Android, Flutter, web, or React Native apps, and can support connecting to a new or existing AWS lambda backend. There is one especially useful feature in Amplify Studio where it can generate UI code for us given Figma mocks. This is something we should look into given our time constraints.

Services that it uses: API Gateway, AppSync, CloudFront, Cognito, DynamoDB, Elasticsearch, Kinesis, Lambda, Lex, Location Service, Pinpoint, Rekognition, and S3.

Based on my reading, Amplify will probably use API Gateway and Lambda for the backend and CloudFront and S3 for the UI, which I will use as my assumption for architecture diagrams.

Pros

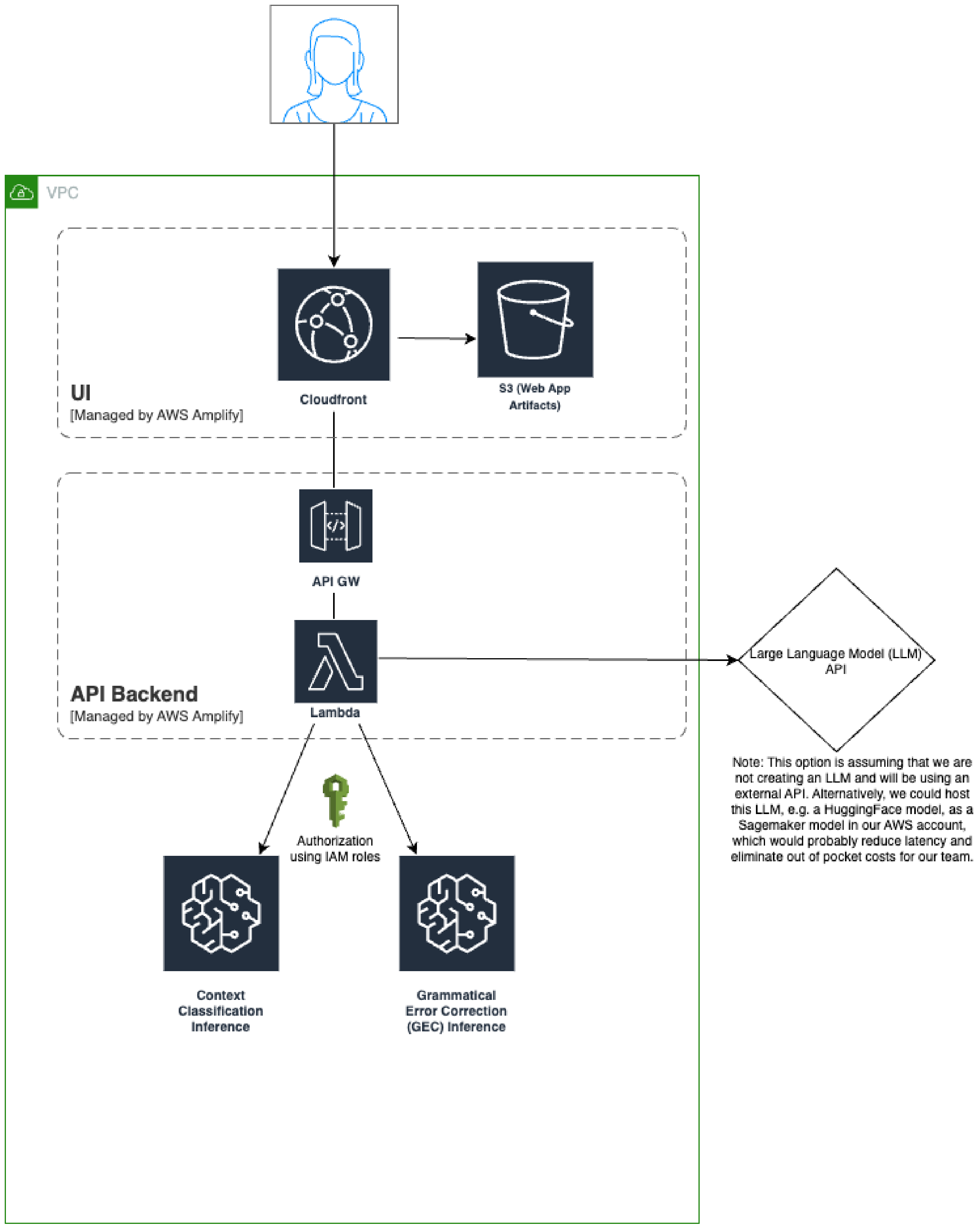
- Easy to set up and use.
- We should be able to get a web app and potentially a mobile app set up easily.
- UI could even be design-to-code (we create Figma mocks, it creates a website).
- Sets up software deployment pipelines in one command.
- Managed service so no need to update EC2s, etc.
- Uses lambda on the backend so we only pay for what we use, compute wise. In general, AWS Amplify itself is free and we will only pay for the AWS resources that it sets up and we actually use.
- Good for our timeboxed project.
- Abstracts away infrastructure so less to worry about.

Cons

- Might be more expensive than setting up our own infrastructure since it abstracts away what is actually being created.
- Abstracts away infrastructure which might set limitations based on what we want to do.

Architecture Options

(Recommended) Option 1: Model Trained Inside AWS, Uses Sagemaker for Training and Inference


VPC
UI
[Managed by AWS Amplify]
Cloudfront
S3 (Web App Artifacts)
API GW
API Backend
[Managed by AWS Amplify]
Lambda
Large Language Model (LLM) API
Note: This option is assuming that we are not creating an LLM and will be using an external API. Alternatively, we could host this LLM, e.g. a HuggingFace model, as a Sagemaker model in our AWS account, which would probably reduce latency and eliminate out of pocket costs for our team.
Authorization using IAM roles
Context Classification Inference
Grammatical Error Correction (GEC) Inference

The easiest option for us would be to use AWS Sagemaker for the entire model development lifecycle. We would use Sagemaker Studio for preparing our data, creating a feature store, hosting Jupyter notebooks, training our models, and deploying our models. There is no need to worry about a deployment pipeline for our models at this point as Sagemaker offers one-click deployments. We will likely be able to use one of Sagemakers pre-built Docker containers for hosting, leaving us with more time to focus on tuning the model.

We would want to pursue this option if we believe that the training is going to be compute expensive and it makes sense for us to train our models in AWS instead of using Weights and Biases or our local computers.

Pros

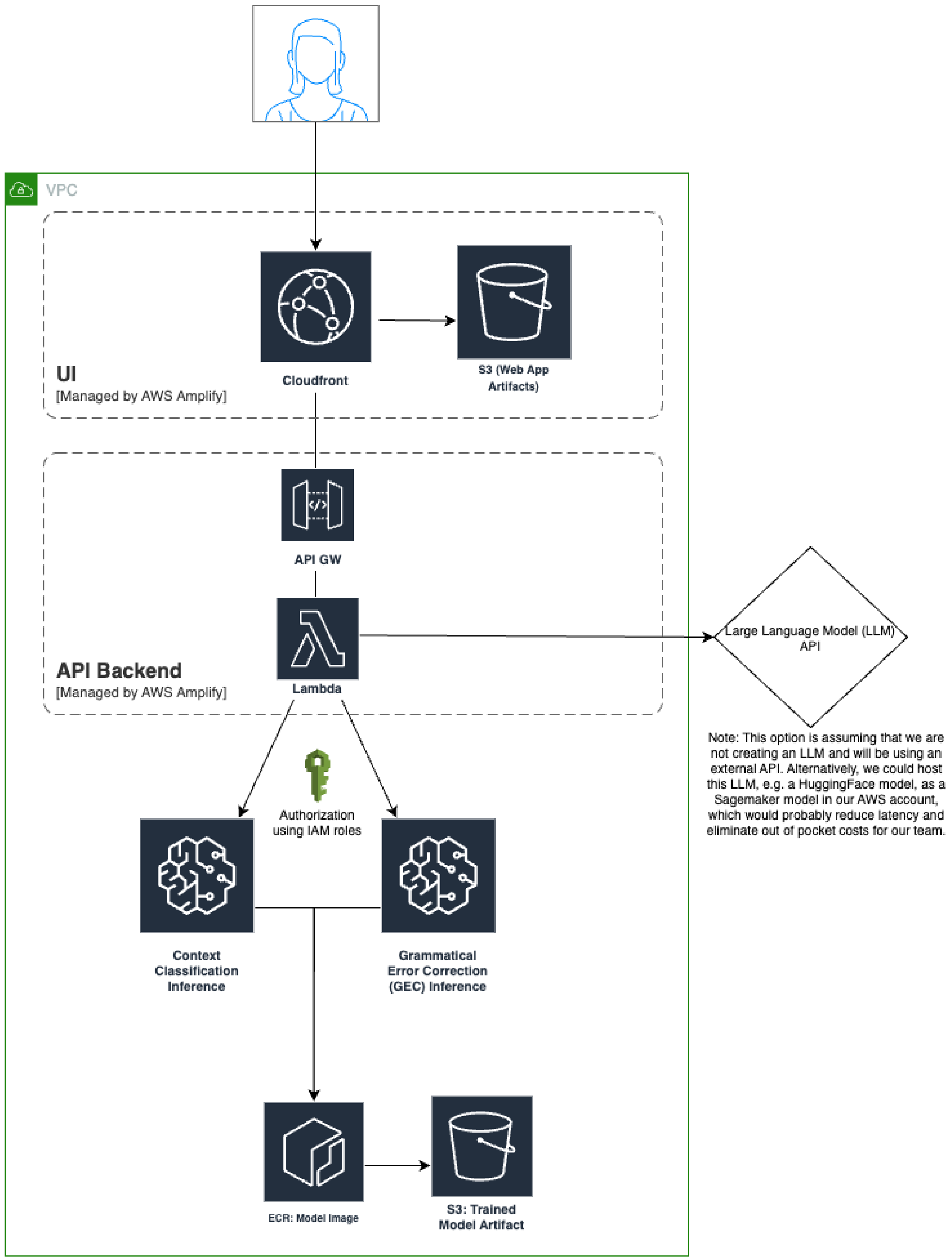
- Easiest approach for creating our models and saves us developer time.
- Gives us an ML pipeline out of the box.

Cons

- Likely more expensive to train on AWS than local or a cheaper website, but we might need to train on AWS anyways depending on how much compute power we need.

Sagemaker offers several options for inference deploys, including serverless inference and real time inference. The differences are in Appendix A. I think we should start with serverless inference to keep our costs down as we do not expect much traffic during capstone, and if we need to scale, we can move to real time inference, which uses an always on EC2.

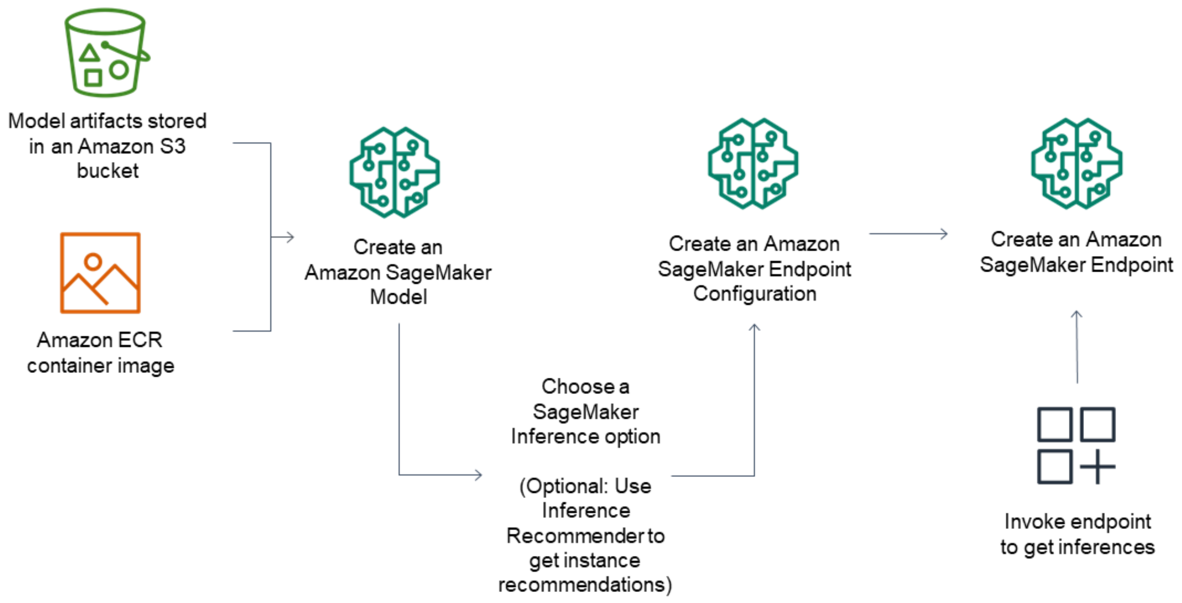
Option 2: Model Trained Outside AWS, Uses Sagemaker for Inference


VPC
UI
[Managed by AWS Amplify]
Cloudfront
S3 (Web App Artifacts)
API GW
API Backend
[Managed by AWS Amplify]
Lambda
Large Language Model (LLM) API
Note: This option is assuming that we are not creating an LLM and will be using an external API. Alternatively, we could host this LLM, e.g. a HuggingFace model, as a Sagemaker model in our AWS account, which would probably reduce latency and eliminate out of pocket costs for our team.
Authorization using IAM roles
Context Classification Inference
Grammatical Error Correction (GEC) Inference
ECR: Model Image
S3: Trained Model Artifact

Option 2 is similar to Option 1 in terms of overall architecture, but instead of using Sagemaker for both training and inference, we would train our model outside of AWS and then upload it to Sagemaker for inference. We can do this using the steps as described in blog post, which I have summarized below:

1. Building the model and saving the artifacts.
    a. Can train the model wherever but once it is trained we need to serialize and load the model as a pickle file into S3: see here for more information.
2. Defining the server and inference code.
    a. When an endpoint is invoked, SageMaker interacts with the Docker container, which runs the inference code for hosting services, processes the request, and returns the response. We will need to create a Flask app for this inference, something like this.
3. Building a SageMaker Container.
    a. Need to create a repository in ECR (Elastic Container Repository) to store the Docker image.
4. Creating Model, Endpoint Configuration, and Endpoint.
    a. All done within Sagemaker. We can use
5. Invoking the model using Lambda with API Gateway trigger.
    a. The lambda would be another Flask app that would be called directly by the UI and orchestrate the model calls and format the response.

This is detailed below and introduces S3 and ECR into our application diagram above.


Model artifacts stored in an Amazon S3 bucket
Amazon ECR container image
Create an Amazon SageMaker Model
Choose a SageMaker Inference option
(Optional: Use Inference Recommender to get instance recommendations)
Create an Amazon SageMaker Endpoint Configuration
Create an Amazon SageMaker Endpoint
Invoke endpoint to get inferences

Pros

- Save on training costs if we can find somewhere else to train our model.

Cons

- Not a one-click model pipeline, there will be some manual steps to zip and upload the file. We could potentially automate this using Code Pipeline so that when someone pushes to the GH repo we upload the pickle file to S3 (using file versioning so that we don't lose old model versions and can "rollback" easily).
- I'm not sure what the benefit is here of using SageMaker to host the models instead of just lambda directly (see Option 3). I'll need to look into this further.

Option 3: Model Trained Outside AWS, No Sagemaker

TODO: More research.

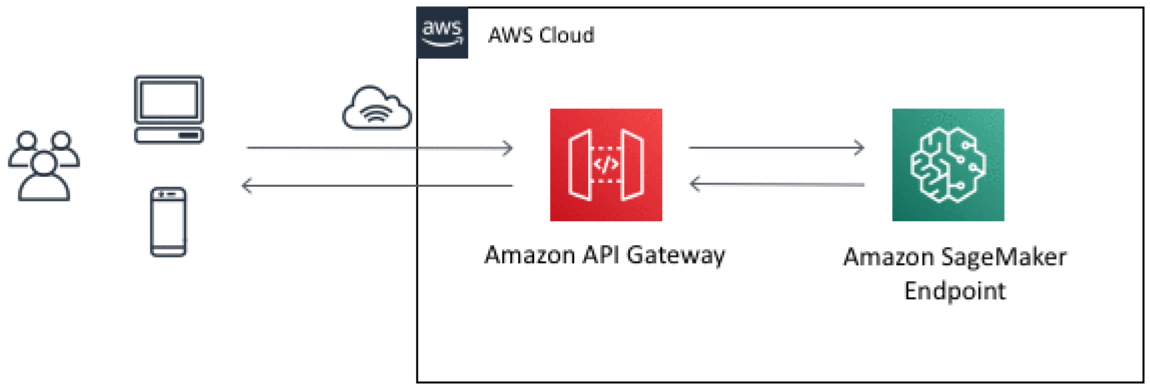
There seems to be a way where we can cut out Sagemaker entirely by just using lambda for hosting our inference. It appears to be similar to Option 2, where the trained model is uploaded to an S3 bucket and we call it using a containerized lambda function. I'm not sure if this will be cheaper than using Sagemaker for hosting our model, or if there are caveats to this approach. I can look into this further if we decide that we do not want to train our model on Sagemaker, i.e. we do not want Option 1.

Resources:

- https://aws.amazon.com/blogs/machine-learning/build-reusable-serverless-inference-functions-for-your-amazon-sagemaker-models-using-aws-lambda-layers-and-containers/
- https://aws.amazon.com/blogs/compute/deploying-machine-learning-models-with-serverless-templates/
- https://aws.amazon.com/blogs/machine-learning/deploying-machine-learning-models-as-serverless-apis/
- https://towardsdatascience.com/saving-95-on-infrastructure-costs-using-aws-lambda-for-scikit-learn-predictions-3ff260a6cd9d (Article title describes that can be a cheaper to use Lambda instead of Sagemaker, but behind a paywall for me)

(Not Recommended) Option 4: Sagemaker Models as APIs, No Lambda

Another architecture that I saw while researching was just hosting the models on SageMaker and calling it directly from the UI via API Gateway. I wanted to call out this architecture as its simplicity is tempting, but I want to avoid having the UI orchestrating these API calls and handling the data processing/decision making (the if/thens that on the sequence diagram). Right now it's not too much logic, but especially when we include personalization and additional model inputs (see P1 in roadmap doc) this logic will become heavy and it'll be better for latency and UX to have this logic on the backend and have the UI handle only making API calls and rendering the response.



Reference:
https://aws.amazon.com/blogs/machine-learning/creating-a-machine-learning-powered-rest-api-with-amazon-api-gateway-mapping-templates-and-amazon-sagemaker/

Pros

- Simple architecture.

Cons

- Potentially more latency/heavy logic on the front-end to orchestrate all the model API calls, can also get messy.
- We will need more APIs endpoints for P1 (login, etc) and so it makes sense to just set up this lambda infrastructure now.

<TODO> Deployment Pipelines

Software Deployment Pipeline

Requirements:

In scope:
- CD so that when you push to the GH repo, it will deploy your changes for both the UI and the API

Out of scope:
- Multiple environments

Model Deployment Pipeline

API Contracts

TODO need to update this contract once I know what model inputs/outputs and an example expected conversation on the front-end look like

From the front-end, all we really want model-wise is to return text, so we can just have one API endpoint that will return the text to display. An example API contract would look like this:

`` ``` ``

Endpoint: GET /text

Headers:

1. contentType: application/json
   // v2 can include an authorization header for user access control, but for now we can just leave this as empty, unless we decide to add personalization. We can do service access control, i.e. who can call the API, using IAM/security groups.

Request body:

{

  “conversationId”: (int) 1, // Options for MVP are 1 or 2

  “stepNumber”: (int) 2, // Conversation step pair we are on

  “attemptNumber”: (int) 1, // TODO use the number of attempts per question to determine the response, if we hit X attempts (3?) then we move onto the next question

  “text”: (string) “Si me gusta voy a la biblioteca.”

  // v2 can include language as a field, but for now we’ll just assume it’s Spanish

}

Response:

1. Code: 200 OK
   Body:
   {
     “correct”: (bool) true/false, // Used to increment the attempt number on the front-end
     “nextStep”: (int) 1, // Can be 1 - 3, 1 = move to next conversation pair, 2 = prompt for errors, 3 = end conversation
     “text”: (string) “Bueno! Que te gusta leer?” // This could either come back from the LLM, if the context was good, or could be one of a few hard-coded responses prompting for a new input from the user. We can store these responses in code right now, an optimization as we expand to different languages would be to store these in a database.
   // errors: list of errors to show user – attempt 1, high level error, attempt 2, more information
   }

2. Code: 401 Unauthorized // This would come from API Gateway/whatever we’re using to control service authorization. If we’re planning on doing personalization, this could also

come back from the application (or potentially an authorization lambda tied to API Gateway, depending on architecture).
Body: {}

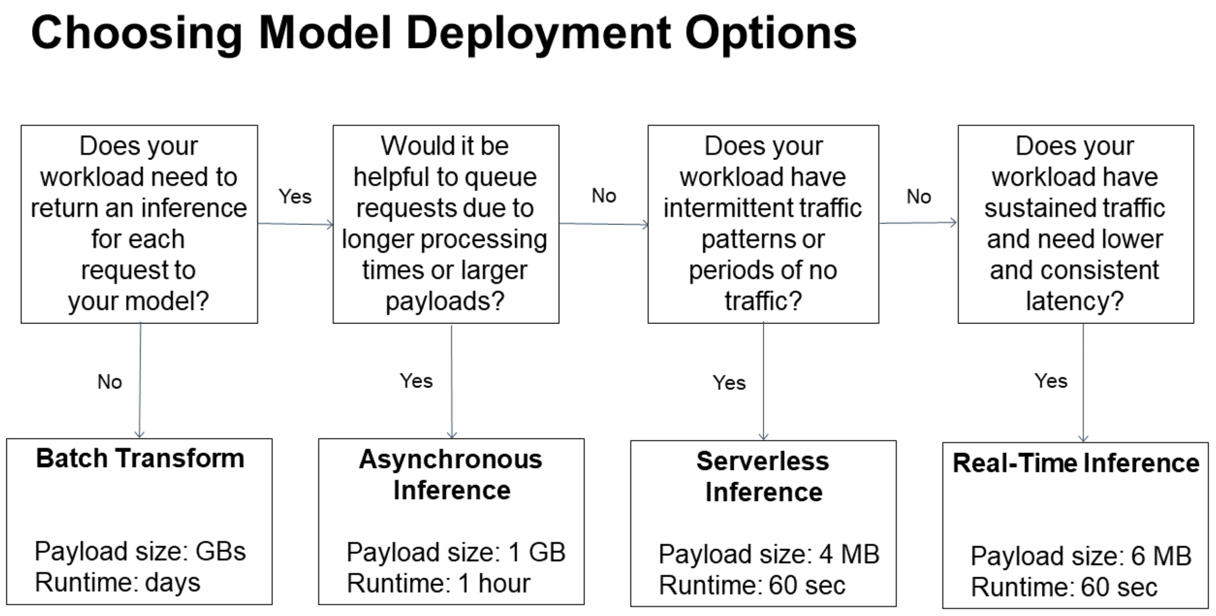
3. Code: 500 Internal Server Error // Server side issues
   Body: {}

```

Small note: it is recommended to encode the responses in UTF-8 in the UI to account for different languages and keyboards. This is the encoding for 98% of websites and means that we don't need any encoding/language related information in the API response.

# Appendix

## Appendix A: Sagemaker Inference Deployment Options


Choosing Model Deployment Options
Does your workload need to return an inference for each request to your model?
Yes
Would it be helpful to queue requests due to longer processing times or larger payloads?
No
Does your workload have intermittent traffic patterns or periods of no traffic?
No
Does your workload have sustained traffic and need lower and consistent latency?
No
Yes
Yes
Yes
Batch Transform
Payload size: GBs
Runtime: days
Asynchronous Inference
Payload size: 1 GB
Runtime: 1 hour
Serverless Inference
Payload size: 4 MB
Runtime: 60 sec
Real-Time Inference
Payload size: 6 MB
Runtime: 60 sec

- Serverless Inference: *Serverless inference* is ideal when you have intermittent or unpredictable traffic patterns. SageMaker manages all of the underlying infrastructure, so there's no need to manage instances or scaling policies. You pay only for what you use and not for idle time. It can support payload sizes up to 4 MB and processing times up to 60 seconds.
```

- Real-Time Inference: *Real-time inference* is ideal for online inferences that have low latency or high throughput requirements. Use real-time inference for a persistent and fully managed endpoint (REST API) that can handle sustained traffic, backed by the instance type of your choice. Real-time inference can support payload sizes up to 6 MB and processing times of 60 seconds.

Reference:
https://docs.aws.amazon.com/sagemaker/latest/dg/deploy-model.html#deploy-model-options