Eccia

Architecture Documentation and Scaling Plan

Chris Roddy
SSW 565

Table of Contents

Introduction

The World Health Organization reports 301 million people meet the diagnostic criteria for an anxiety disorder. Having personally experienced the impact of disordered anxiety, I understand the challenges it brings to daily life. Mental health challenges have been steadily increasing across the United States since the mid 2000's — anxiety is the most common family of mental health disorders, but only 27.6% of people will receive treatment. This statistic is shockingly low, and compelled me that change was necessary.

Cord is a GPT powered emotional support chatbot designed to support people living with anxiety, and is now almost a year old; the project began in January 2024 and has continued to present day. The system is implemented using React Native, Flask, MongoDB, and OpenAI's API. Only a development build of Cord currently exists — to reach the tens of millions of people with anxiety disorders in the United States, the system must transition to a production environment and deploy on a scalable cloud provider. Additionally, the current lack of documentation makes it challenging for developers to join this project in the future without significant onboarding investment.

This document overviews Cord's architecture, quality attributes, risk analysis and mitigations, deployment plan, architectural proposals, and future product expansion. Additional information on Cord's background, stakeholders, user-centered research, system requirements, business strategy, and high-level solution outline can be found in an abridged version of the original proposal and domain analysis.

Quality Attribute Scenarios

Attribute	Security
Source	Privileged User
Stimulus	Access End User Data
Environment	All operating conditions
Artifact	Server
Response	The system shall verify the identity of any person attempting to access user data
Measure	Users attempting to access server data must be reauthorized and validated for data access

In the context of software systems handling potentially sensitive user-data, like a mental health support chatbot, there are increased requirements to remain compliant with patient privacy laws such as HIPAA. This makes any security vulnerabilities that may lie within the system a significant legal risk for the system owner — security must remain at the forefront of thought when making design decisions. Any additions and changes must be evaluated for potential security risk.

Attribute	Availability
Source	End User
Stimulus	Message Chatbot
Environment	All operating conditions
Artifact	Client, Server
Response	End user should be able to message and receive a response from the chatbot provided they have a stable internet connection
Measure	System should be highly available (99.999% uptime)

For any software system to retain a consistent userbase, it must prove to be a reliable service — downtime must be minimal. This is magnified when considering the system is designed as a mental health support tool; for the system to be widely adopted and trusted by patients and providers, it must be reliable and available.

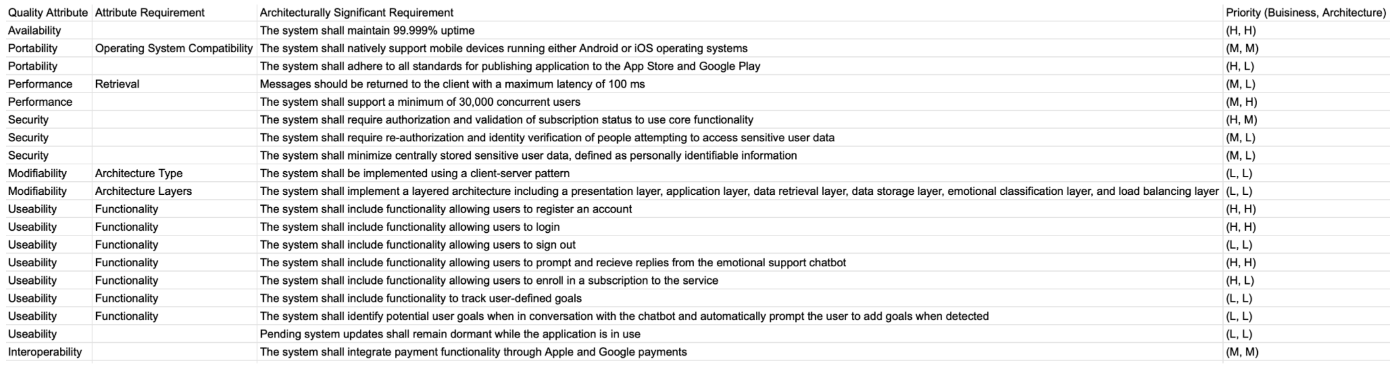
Attribute	Performance
Source	End User
Stimulus	Message Chatbot
Environment	Peak traffic
Artifact	Client, Server
Response	The system shall return responses promptly
Measure	Messages should be returned to the client with a maximum latency of 100 ms

A latency of 100 ms, while not instantaneous, is generally seen as the standard for software. There is minimal business risk associated with this quality attribute, as 100 ms is attainable by most software applications, even those without implemented performance optimizations.

Attribute	Cost Efficiency
Source	System Owner
Stimulus	Third-Party GenAI Token Generation
Environment	All operating conditions
Artifact	Server
Response	The system shall minimize third-party API costs
Measure	Input/Output token cost should not exceed $12.50 per million tokens

The pricing scheme described above aligns with the OpenAI's standard model pricing for gpt-4o. Uncached input tokens cost $2.50/million, while output tokens cost $10/million. Cached input tokens cost $1.25/million — this suggests opportunity for future cost optimizations. Business risk associated with this quality attribute rises exponentially as the userbase scales. The more users actively use the system, the more prompts are processed by OpenAI. This results in higher cost — which will be mitigated by subscription fees.

Utility Tree



Quality Attribute	Attribute Requirement	Architecturally Significant Requirement	Priority (Buisiness, Architecture)
Availability		The system shall maintain 99.999% uptime	(H, H)
Portability	Operating System Compatibility	The system shall natively support mobile devices running either Android or iOS operating systems	(M, M)
Portability		The system shall adhere to all standards for publishing application to the App Store and Google Play	(H, L)
Performance	Retrieval	Messages should be returned to the client with a maximum latency of 100 ms	(M, L)
Performance		The system shall support a minimum of 30,000 concurrent users	(M, H)
Security		The system shall require authorization and validation of subscription status to use core functionality	(H, M)
Security		The system shall require re-authorization and identity verification of people attempting to access sensitive user data	(M, L)
Security		The system shall minimize centrally stored sensitive user data, defined as personally identifiable information	(M, L)
Modifiability	Architecture Type	The system shall be implemented using a client-server pattern	(L, L)
Modifiability	Architecture Layers	The system shall implement a layered architecture including a presentation layer, application layer, data retrieval layer, data storage layer, emotional classification layer, and load balancing layer	(L, L)
Useability	Functionality	The system shall include functionality allowing users to register an account	(H, H)
Useability	Functionality	The system shall include functionality allowing users to login	(H, H)
Useability	Functionality	The system shall include functionality allowing users to sign out	(L, L)
Useability	Functionality	The system shall include functionality allowing users to prompt and recieve replies from the emotional support chatbot	(H, H)
Useability	Functionality	The system shall include functionality allowing users to enroll in a subscription to the service	(H, L)
Useability	Functionality	The system shall include functionality to track user-defined goals	(L, L)
Useability	Functionality	The system shall identify potential user goals when in conversation with the chatbot and automatically prompt the user to add goals when detected	(L, L)
Useability		Pending system updates shall remain dormant while the application is in use	(L, L)
Interoperability		The system shall integrate payment functionality through Apple and Google payments	(M, M)

The utility tree pictured above captures 19 architecturally significant requirements. The majority of attribute requirements center around usability — this reflects Cord's design philosophy as a user-centered product over a purely technical endeavor. The end objective of developing this application is launching to market as a self-sustaining, viable SaaS product.

Architecture

Cord's architecture adheres closely to a traditional client-server pattern, and this has proven to be an effective design thus far. However, Cord's architecture faces two main challenges — access to the application is restricted to a development build, and the ability of the server to handle multiple clients simultaneously has not been verified. The system's architecture must be refactored for scalability. A cloud deployment is the most effective way to transition Cord into a production environment. Discussion on cloud deployment details can be found under Physical View.

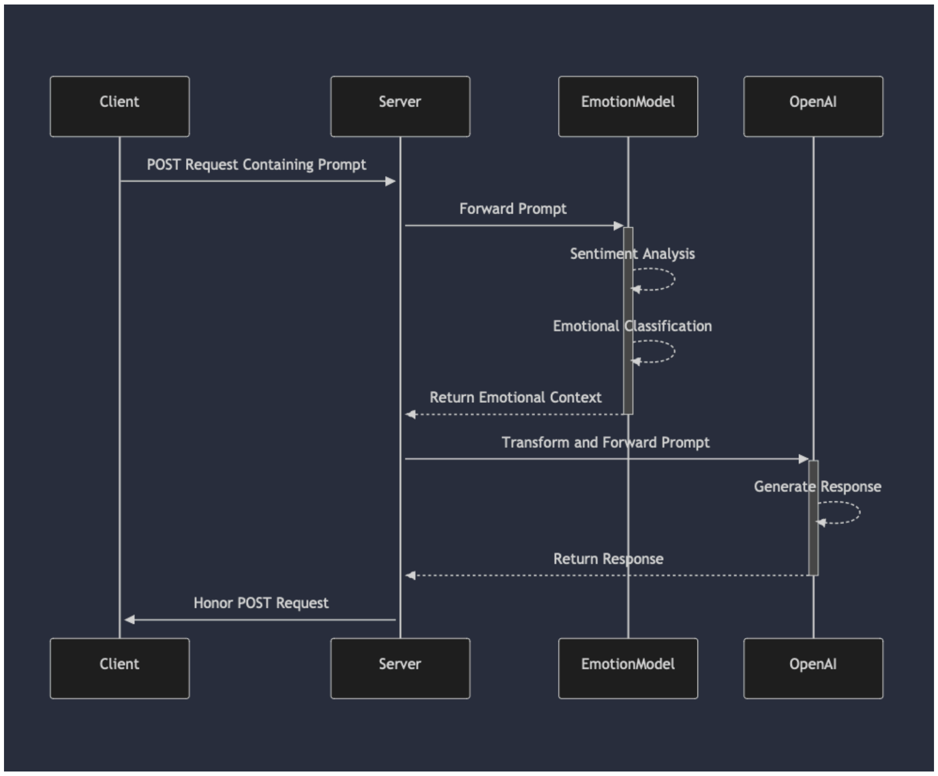
Emotional Classification

While reading Marc Blackburn's *Permission to Feel*, I learned that infants aren't born with the ability to detect emotions in others; their ability to recognize emotion is limited to "good" and "bad", which can be reframed as "positive" or "negative". Humans gradually develop the ability to detect emotion — precision increases as we grow older. First we learn to recognize "happy" and "sad", then "fear", "anger", disgust", and so on. This reminded me of sentiment analysis in artificial intelligence, and gave me an idea — could I incorporate a combined sentiment analysis and emotional classification processing layer into Cord?

Feeding the output of sentiment analysis into specialized emotional classification models for “positive” emotions (happy, calm, content, etc.) or “negative” emotions (sad, anger, disgust, fear, etc.) would mimic how humans develop from a developmental psychology perspective — I hypothesize that adding this extra layer would improve model accuracy compared to using a generalized emotional classification model like this one found on huggingface. Adding pre-processed emotional context to a user’s prompts before calling OpenAI’s API has the potential to significantly increase usability. GPT’s outputs may be of higher quality and of greater empathy than if the context was not added, though this remains to be verified. It must be noted that incorporating emotional classification as another layer to the architecture has the potential drawback of significantly impacting performance. The architectural documentation following assumes successful integration of emotional classification without significant consequences to performance.

Process View

Below is a sequence diagram for the most important functionality of the system — the ability to communicate with the chatbot:



The client communicates with the server through a POST request implemented as follows:

```
const response = await fetch("http://127.0.0.1:5000/chat", {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({
            message: input,
            memory: updatedUserChats,
          }),
        });
```

It is worth discussing the body parameters passed by the client's outbound request. The user's prompt is assigned to the message key, but the locally stored chatlog is also passed in the request. This is because OpenAI's API, as currently integrated within the system, does not keep context — prompt completions are isolated events. Passing the chatlog serves as a way to bypass this limitation at little additional cost. Optimizations can be implemented to further reduce this added "memory" cost, such as removing the chatbot's responses from the user chatlog. This optimization is currently being worked on, though enabling this functionality has led to unexpected behavior. The exact cause is still being investigated. The system then receives this incoming request from the client, and processes it as follows:

```
@app.route("/chat", methods=["POST"])
def chat():
    memory = []

    for item in request.get_json()["memory"]:
        sys_prompt = {
            "role": "system",
            "content": item["type"] + ": " + item["text"]
        }

    memory.append(sys_prompt)

    return answer_prompt(memory, request.get_json()["message"])
```

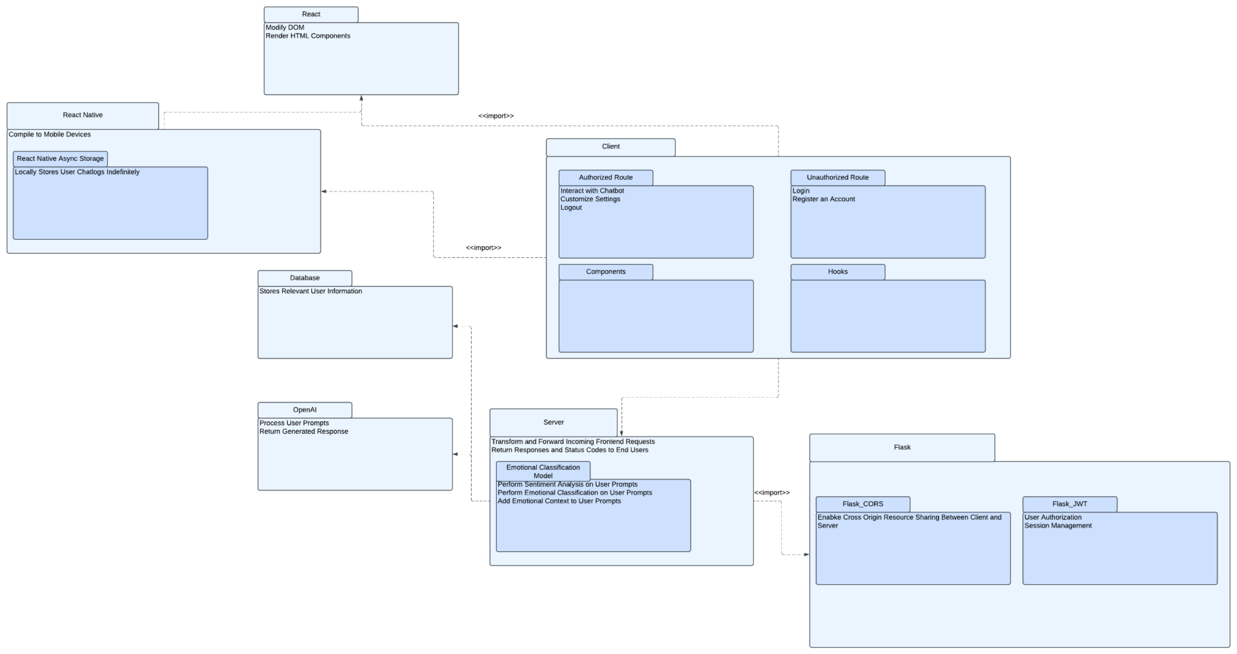
The system's current method of transforming the chatlog into OpenAI friendly formats has an O(n) time complexity — the amount of time it takes to process a single request increases linearly the longer a conversation is active. Additionally, the cost to process also increases linearly. A straightforward strategy to reduce this processing bottleneck is to limit the amount of context the system receives. The mitigation discussed above, removing chatbot responses, is a first step. Afterward, limiting the section of the chatlog initially passed to the server from the client to a fixed number of prior prompts. This aids in the scalability of the system by reducing processing load. Techniques to transform the chatlog into system prompts without using a loop can also be explored.

```
from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(
    model="model name"
    messages=[
        {
            "role": "system" | "user"
            "content": "Prompt"
        }
    ]
```

The code listed above is the basic structure of using OpenAI's API for basic prompt completion with a GPT instance. Specific system prompts used by Cord have not been included in this documentation due to frequent changes. When performing a basic completion like the one above, the chat completion functionality expects a model name, and an array of message objects using in the format specified in the example code.

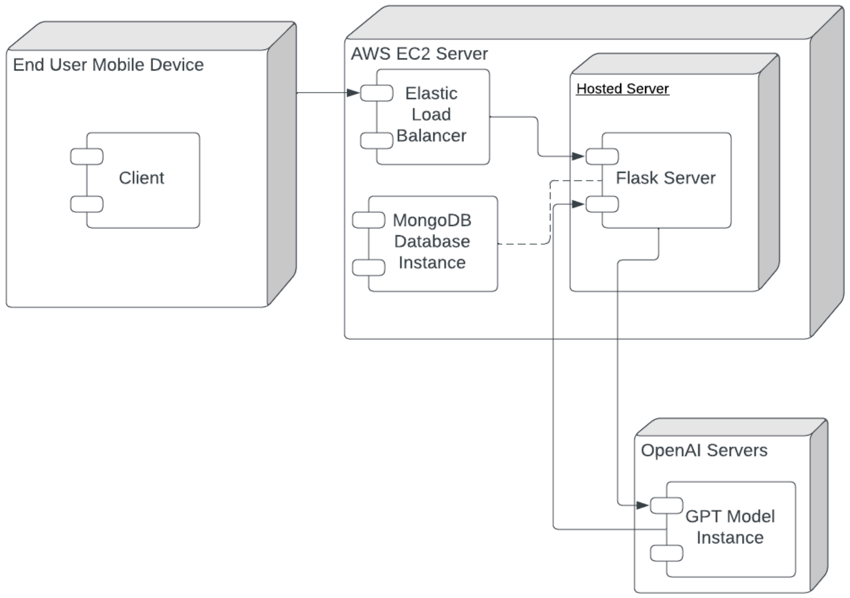
Development View

A package diagram of the project’s file and dependency structure is pictured below — this is condensed for clarity, and represents the most critical subsystem file structures and external interfaces:


React
Modify DOM
Render HTML Components
React Native
Compile to Mobile Devices
React Native Async Storage
Locally Stores User Chatlogs Indefinitely
<<import>>
<<import>>
Client
Authorized Route
Interact with Chatbot
Customize Settings
Logout
Unauthorized Route
Login
Register an Account
Components
Hooks
Database
Stores Relevant User Information
OpenAI
Process User Prompts
Return Generated Response
Server
Transform and Forward Incoming Frontend Requests
Return Responses and Status Codes to End Users
Emotional Classification Model
Perform Sentiment Analysis on User Prompts
Perform Emotional Classification on User Prompts
Add Emotional Context to User Prompts
<<import>>
Flask
Flask_CORS
Enabke Cross Origin Resource Sharing Between Client and Server
Flask_JWT
User Authorization
Session Management

Within the client lies an app folder containing files associated with the unauthorized route; the “main menu” file allowing the end user to navigate to the sign-in or register page, represented by respective typescript component files. In a subfolder within the app folder lies files rendering pages on the authorized route — the main chatbot screen is rendered from “index.tsx”, settings page by “settings.tsx”, and a layout control file dictating which page gets rendered. The client is simultaneously dependent on React and React Native. React Native allows native rendering through provided HTML-like components, and certain modules within the client make use of React’s hooks like useEffect() to coordinate with external dependencies.The server exists as a single folder with all of the files contained within, containing the Flask server, MongoDB interface, and the script communicating with OpenAI.

Physical View



Cord is not currently deployed to the cloud, and the deployment diagram pictured above is for the eventual planned public release of the system. AWS EC2 is the preferred option for deployment to a production environment for several reasons — the first is that AWS is already integrated into the tech stack as the cloud provider for the system’s MongoDB instance. Consistency in cloud provider is valued, as this will improve system cohesion. Another reason is convenient implementation of load balancing as a scaling tactic — the AWS Elastic Load Balancer already exists within Amazon’s established technical ecosystem, so integration should be seamless.

Conclusion

The winding road leading to Cord’s launch inches closer to the finish line each day. The next major hurdle for Cord to overcome is the deployment to a production environment so the system can be tested by real users — the development of this application hasn’t been without challenge, and has not yet fully resolved, but Cord’s future is bright with clear skies ahead. The question is not whether this application *will* release to app stores, but rather, *when*.