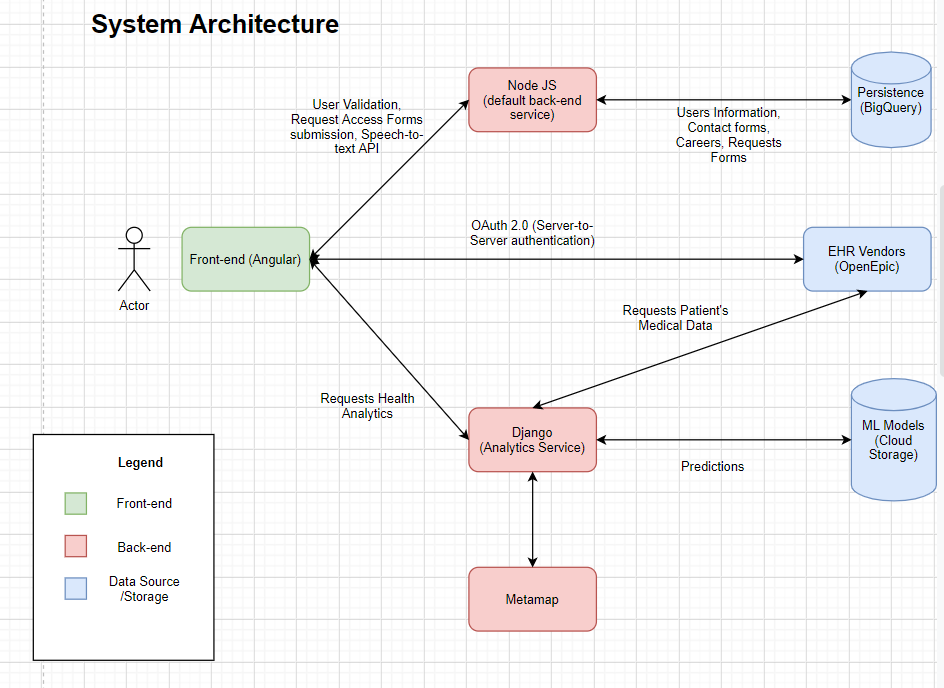
**System Architecture:**



The diagram depicts all the components currently in production or in development at ScriptChain.

**Front-end:** The front-end of the application is entirely designed in Angular. The front-end makes relevant calls to NodeJS and Django server endpoints for any server related processes like- user authentication, generate analytics.

Also, another important function of the front-end is to authenticate the ScriptChain platform to request resources from EHR vendors.

**Back-end:** Currently we have two backend services deployed in production:

1. **NodeJS:** The node server handles all the application/user level functions like authentication, sending acknowledgment emails, storing requests from careers page, etc. A list of endpoints in this server can be found in the backend endpoints document.
2. **Django:** The django server mainly relays back the health analytics of a patient from the models deployed by the Machine Learning team. The django server also makes http requests to the EHR vendors to fetch the patient data.

**Django Service:** Django follows a Model-View architecture, where the models depicts tables in the database, which in our case is bigquery. Since, our purpose is to create a Restful API in django, Django Rest Framework was used to create the API. All the endpoints are in the **views.py** file. The endpoint urls can be found in the urls.py file. Right now we just have two endpoints:

1. “/”: This is an echo endpoint. You should get “Hello, world!” as a response when you hit this endpoint from your browser.
2. “/generateAnalytics”: This endpoint is used to generateAnalytics from the patient data. This endpoint accepts post requests.. An easier way to test this endpoint is to use postman.

**Scripts for EHRInterface:** Our architecture would be more modular if we maintain the EHR related operation in Django. However, performance could be a trade-off. All the scripts to fetch the patient data for CDV risk analysis are in the EHRInterface package. There are individual modules for writing scripts for each EHR. Right now we have implemented the code to get the data from Epic.

**Test Suite in Django:** Wrote a test script to test the response of generateAnalytics/ endpoint in tests.py

**Manage.py:** This file is the driver file for django. To start the django server locally use this file. Command to start the server: python3 manage.py runserver.

**Django Deployment:** The default backend service is Node, and it’s maintained in GCP AppEngine with the same name. But, the Django backend service is named as analytics.You can track the resource usage of this service by choosing this service in the home page of AppEngine.

For deployment, we use the analytics-app.yaml file to specify the configuration relevant to the service. Although django ships with a default web-server we cannot use it in production. We use the gunicorn web-server to handle requests to the django service.

The routing configuration for this service is maintained in dispatch.yaml file under the directory of the default service, which is Node.

**Persistence Components:**

**BigQuery**: All the application specific data, like- User profiles, resumes of job applicants, etc are stored in ScriptChainProd project in GCP.

**Cloud Storage Bucket**: This has not been done yet, but it would be efficient to deploy all the Machine Learning models to the cloud storage bucket in serialized form. You can unpack and use them in Django to generate predictions.

**Note:** For a more comprehensive view of the System Architecture you can refer to the charts in <https://drive.google.com/file/d/1OpAASiCAupkD9pg-_KyUeQHfktStV5Y_/view?usp=sharing>.

**Software Development Functionalities:**

* **Display CDV risk analysis predictions:**

**Problem Statement:** Collect relevant information of a patient from the EHR vendor and generate predictions from trained models yet to be deployed in Cloud Storage Bucket.

**Input:** Expect the provider to search for a patient using the name/ ID/ email/ activity/ appointment.

**Output:** Collect the relevant medical information of a patient from anEHR vendor,

Unpack the relevant ML model yet to be stored in Cloud Storage Bucket, and generate predictions.

**Workflow:** Angular requests Django.

* **Speech-to-text conversion:**

**Problem Statement:** Choose an API that converts speech to text, and send a request to the EHR vendor to save the notes.

**Input:** Provider’sSpeech.

**Expected** **Output:** Textual representation of provider’s speech to be saved in the EHR portal.

**Workflow:** Django/Node requests Speech API, then Django/Node sends a request to the EHR vendor to save the text.

**Trade-offs:** We can get better packages in python that do the conversion, but there’d be some latency in the application. On the other hand, a javascript API might not be as accurate, but would perform better in terms of performance.

* **Billing and Coding:**

**Problem Statement:** To identify the billable keywords from clinical notes.

**Input:** Corpus of text sourced from an EHR vendor or through the speech recognition API yet to be integrated with our system.

**Expected** **Output:** Identify the billable keywords from text and send a request to the EHR vendor to store the billable data.

* **Metamap Deployment:**

**Problem Statement:** One of the prediction models uses an external tool to generate lower-dimensional mappings of clinical texts, i.e., it extracts the important keywords out the clinical text. The extracted keywords are given as input to the model. But, the problem is to deploy and use the external tool in our environment.

**Solution:** Upload the file to the App Engine server. There are several methods to this. The naive approach is to upload the file directly from the local machine using gcloud sdk or by adding the file to the github repo, but the issue is that the file is too large to commit to a remote repo or to upload it to the server.

Another approach would be to upload the file to dockerhub and download it during build. **This might work.**

If the approaches don’t work, another approach would be to make http requests to the metamap server to generate the mappings.

**Readmission prediction model:**

We choose the ClinicalBert language model for our downstream task. The clinicalbert model is available both in the Huggingface transformers library. An example of our downstream can be found at-

1. <https://github.com/EmilyAlsentzer/clinicalBERT>
2. <https://github.com/kexinhuang12345/clinicalBERT>

Unlike normal BERT the clinicalbert model knows the patterns in clinical notes. Our working copies of clinicalbert can be found at the following links-

<https://colab.research.google.com/drive/1B94FNCm0mmJb6sIPQ7wzcI37uoNKhvhd>

<https://colab.research.google.com/drive/1VTPKLjj2tUBhHuhaFJSdt0Zkg9qW_U0f?usp=sharing>

<https://colab.research.google.com/drive/1oRt2B6zRMn9QE3pst9LfQG3IMEZJybNX?usp=sharing>