2019

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

METAFACE: A SYSTEM FOR BENCHMARKING FACE PROCESSING API'S

Kevin Gay

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management
University of North Carolina Wilmington

2019

Approved by
Advisory Committee

Dr. Lucas Layman	Dr. Jefferey Cummings

Dr. Karl Ricanek, Chair

Accepted By

Dean, Graduate School

Table of Contents

ABSTRACT

MetaFace: A System for Benchmarking Face Processing API's. Gay, Kevin, 2019. Capstone Paper, University of North Carolina Wilmington.

Even though APIs and facial processing are new areas of technology, there are already an abundance of cloud-based APIs that provide facial processing services to companies, researchers, and government agencies. Google, Microsoft, IBM, and Amazon are some of the major companies that offer these services; however, there are dozens of other companies out there, from startups to established multinationals, that offer the same services. Many of the lesser known companies may be cheaper and provide more optimal solutions, i.e. better measured performance, however they lack the name recognition of the software giants. Because of the dominance of these software giants, i.e. Microsoft, Google, IBM, and Amazon, face processing has received negative coverage in recent news due to lack of performance, gender bias, race bias, and privacy concerns [60, 61, 62]. In some cases, the lesser known entities may have developed algorithms that outperform the giant's, but their collective voices are muted due to the press coverage around the giants. For this reason, I developed a meta-API which calls several face processing APIs, aggregates the outputs, and delivers the outputs in a user configured manner. An end user could use this system to evaluate several solutions against the same set of inputs, performing deep evaluations on against a set of solutions, which currently

does not exist[1]. An end user could also aggregate the results based on the solution that performs the best on a given output to mitigate against algorithm bias or, simply, to generate the most accurate result.

[1] Academic researchers typically perform evaluations on public algorithms. Commercial systems, e.g. Amazon Rekognition, may not make public their full solution for face processing. Hence, this solution could be used by academics as well as private corporations to determine the efficacy of said solutions on their use-cases. Today, many of these companies provide performance data on proprietary datasets which showcase their solution in the most favorable light.

LIST OF FIGURES

Figure Page

CHAPTER 1: INTRODUCTION

Application Programming Interfaces (APIs) and face processing are two relatively new areas of technology. A large portion of the Internet communicates through APIs [65]. Companies use private APIs to access their onsite data and functionality through the simplified and secure layer that the API provides. Companies also use public APIs to access data and functionality that is offered by other providers. They have become so desirable that thousands of companies have started as API companies, which offer only APIs as products. Face processing is another buzzword in the modern computing industry, and it has made many headlines in recent years with regards to security, access control, and research. More recently, however, companies have merged the two technologies to create face processing APIs.

The multi-billion-dollar cloud company, Red Hat, defines an API as "a set of tools, definitions, and protocols for building and integrating application software" [44]. What about APIs make them so desirable that they have become a billion-dollar industry? APIs have the ability to wrap around an extremely complex system and expose data and functions in a way that outside developers can consume into their own applications. They provide organization, communication, and security to applications that are inside or outside of a business that develops them.

Face processing is a billion-dollar industry [64] that incorporates three distinct operations. The first operation is face recognition, which is the ability to match a face image with a set of known faces maintained in a database. Face analytics, the second operation, derives information about a person solely from their face. These attributes might include age, gender, ethnicity, eye color, and may even be able to tell whether

someone has diabetes. The final operation is face attributes, which provide characteristics about a face in an image that can be altered from one image to the next. These include complexion and presence of eyewear, scars, makeup, tattoos, and more. Face processing is already being used in several applications today, and more applications are still being discovered, tested, and implemented. Face recognition can be used to identify missing children, flag dangerous individuals at airports, allow a person access to their phone, and automatically tag a person in an image on social media. Face analytics and attributes can be used to provide targeted ads to an individual at a gas station based on their age and gender. More recently, face analytics is being extended to detect whether an individual is afflicted with diseases based on a high definition image of their face. Face attributes can also be used to identify individuals, particularly if they have a unique scar or tattoo on their face. For the current project, the area of focus is face analytics, particularly predicting age and gender.

Figure 1 - Use cases of face processing [63]

Face processing APIs were created because face recognition historically required expensive software suites that were difficult to maintain. They had to be integrated into

entire camera systems and supported with even more expensive hardware. When transformed into APIs, any system can utilize the technology if it has Internet access. Furthermore, all of the computation is done in the cloud, so the expensive computers that run the algorithms can be supplied by another company. Most of the readily available face APIs can handle thousands or millions of requests at any given second, so there is very little risk of downtime. This makes them well suited for real-time image and video analysis, which fulfils a variety of use cases (see Figure 1).

Even though face APIs are new, there are already hundreds of companies that offer them from billion-dollar companies, like Amazon, to smaller companies, like Sighthound. All of these companies claim that they have the superior face processing algorithms, but, surprisingly, there are no public comparisons about how each company's algorithm performs when presented with the same images. A goal of the current project is to understand the variations in algorithm performance across these APIs.

Before beginning this project, an initial search was conducted to find comparisons between existing face APIs. The only comparisons found existed either on blogs or on a particular face processing company's website (see Figures 2-4 for blog post examples). The comparisons only note the specific services that each API provides or vague results as an unexplained percentage. No unbiased comparisons were found that measure each API's performance on a specific dataset. Furthermore, there was no mention of how each API performs on different characteristics like age, race, gender, or any combination of the different metrics. It is important to realize that the algorithms that these companies offer are trained on datasets containing hundreds of thousands of images. However, little is

known of what images are contained within these datasets (e.g., all images could be of males between the age of 30 and 40). One primary purpose of this project is to see how well these algorithms perform on images of varying ages and genders.

	Kairos	Amazon	Google	Microsoft	IBM	Affectiva	OpenCV
Face Detection	✓	✓	✓	✓	✓	✓	✓
Face Recognition (Image)	✓	✓	✗	✓	✗	✗	✗
Face Recognition (Video)	✓	✓	✗	✗	✗	✗	✓
Emotional Depth (%)	✓	✗	✗	✓	✗	✓	✗
Emotions Present (Y/N)	✓	✓	✓	✓	✗	✓	✗
Age & Gender	✓	✓	✗	✓	✓	✓	✗
Multi-face Tracking	✓	✓	✓	✓	✓	✓	✓
SDK	✓	✗	✗	✗	✗	✓	✓
API	✓	✓	✓	✓	✓	✓	✗
Ethnicity	✓	✗	✗	✗	✗	✗	✗

Figure 2 - Face API comparison from a Kairos blog post in 2017 [45]

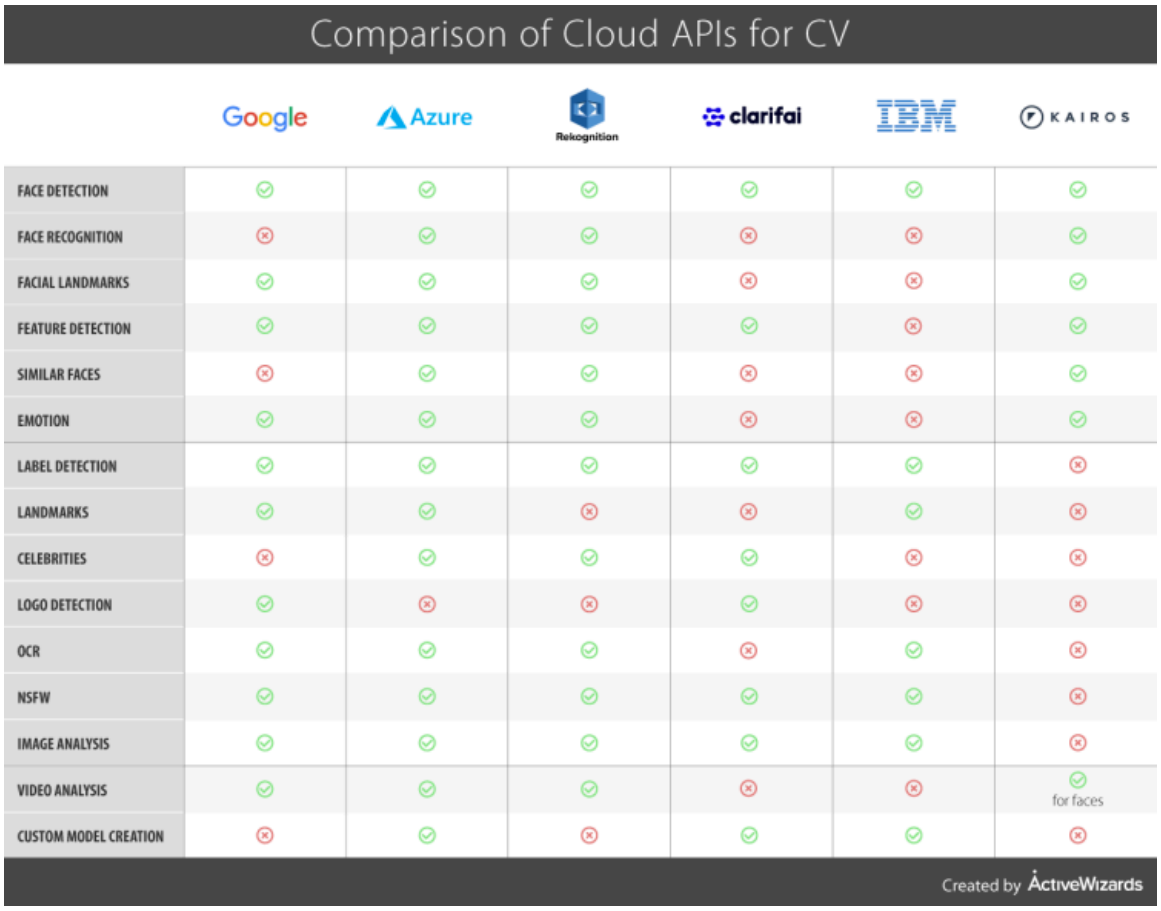


Figure 3 - Face API comparison from datasciencecentral.com blog post in 2018 [46]

API	Available Endpoints	Created by	All Time Users on RapidAPI (as of 5/23/2018)	Pricing
Animetrics Face Recognition	25	Animetrics	10,188	Free up to 1,000 calls/month. Additional pricing tiers available
AWSRekognition	9	Amazon	21,252	Free up to first 1,000 minutes/month for first year.
EmoVu	2	Eyeris	8640	Free up to 500 calls/month. Additional pricing tiers available
Face Recognition and Face Detection	7	Lambda Labs	60,624	Free up to 1,000 calls/month. Additional pricing tiers available
Face++	2	Face++	231,444	Free version available with pas as you go. Additional pricing tiers available
Google Cloud Vision	9	Google	9677	First 1000 units/month are free, then increases depending on features
IBM Watson	7	IBM	2521	Free and premium tiers
Kairos	7	Kairos	12,096	Starting at $500/month. Additional pricing tiers available
Microsoft Face API	30	Microsoft	121,259	Free version up to 20 transactions per minute and 30,000 transactions per month. Additional pricing tiers available
Trueface.ai	9	Trueface.ai	432	Free up to 1,000 calls/month Additional pricing tiers. available

Figure 4 - Face API comparison from blog.rapidapi.com post in 2019 [47]

In the current project, an API suite, called MetaFace, is created that requests the face processing results from several companies and aggregates them into a single API response. Currently, if a researcher wanted to get responses from several face API companies today, they would have to go to each company's website, register, obtain an API key, figure out how to send information to the API, and figure out how to consume the results that are returned as a response. Substantial resourcing would be involved in aggregating a couple of web face processing APIs. Using the MetaFace API, a consumer

only needs to obtain a single API key, pass in a public image URL, and parse a single response. This saves a considerable amount of time in setting up an application or experiment. The API that requests and aggregates results from several third-party APIs is called Gladiator. As a result of Gladiator's creation, datasets of results from the third-party APIs could be created using public face image datasets. Thus, the Bench API was created to tell how each face API company performs on different characteristics.

It is important to realize that each algorithm will differ in performance when presented with different ethnicities, ages, genders, and the various combinations between these input characteristics. Thus, another use case of this service will be the ability to benchmark each algorithm and present which company provides the best solution for a given set of attributes, i.e. range of ages, ethnic group, etc. Not only can the service be used as a way to compare two company's performance, it can be used to track each company's performance over time. It is public information that each API stores the images that its users pass in so that it can be used as input to train their machine learning models in the future. Because of this, each company's performance should get better over time; however, there is not a marketed solution that compares each company without bias. Tracking the rate of improvement could provide beneficial information to a competitor or user of the recognition services, which defines another use case of MetaFace.

For example, Amazon's Rekognition service is offered in different countries, regions, and availability zones, and they may want to purchase information about how their service performs against ethnicities non-native to a given region. For instance, how

well does the Rekognition service in China perform on American faces compared to the Rekognition service in western Europe? This is a similar situation to years ago when there was no reliable way to compare two different pieces of computer hardware, such as CPUs or storage devices.

And finally, a business case can be developed around this work. The Meta-Face API will outperform any single solution, as it is an intelligent aggregate of the top performing solutions. Furthermore, margins will grow as more people adopt this solution because of reduction in cost when several calls are made to each API. Some price points are as low as $0.01 per 100 or 1000 calls. Charging $0.10 per 100 calls provides a significant margin and one interesting business case.

To get a good idea of what different face API companies have to offer, services from small companies to software giants were chosen to be included in this project. These companies include Amazon, Microsoft, IBM, Sighthound, Kairos, and Lapetus. Amazon's Rekognition service is advertised as image and video analysis, as they provide identification of objects, text, inappropriate content, as well as faces. As with most of the face processing APIs, they provide no accuracy metrics for any of their services. They only mention their features and how simple it is to integrate image recognition into your application [48]. Microsoft Azure provides a service known as Face API, and they too provide face processing services from the cloud. Like Amazon Rekognition, they provide a few images for demonstration that they picked out, but they provide no accuracy metrics [49]. IBM provides Watson Visual Recognition for identifying face attributes [50]. Amazon, Microsoft, and IBM are all billion-dollar companies; however, there are

several smaller companies that offer face processing products as well. Sighthound is one of these smaller companies that offers vehicle detection and face detection APIs, but they also have no accuracy data [51]. Kairos differs from the other companies in that they only provide face-related APIs, and they actually provide a chart to compare themselves to the other companies, which was shown in one of the examples above. Like the other companies, they lack data to back up how well their algorithm performs in comparison [52]. Lapetus is the final company that was included in the project. They offer an API called Chronos, which derives age, gender, BMI, and whether a subject is a smoker or non-smoker all from a single image. Lapetus actually mentions how their algorithms are built and what data their models are trained on but have no comparison of how they perform against the other companies [53]. One hypothesis is that no comparisons can be found online because someone hasn't put in the manual work and research to do so. Gladiator can help fill this void since is a tool that can provide results from different services. This project aims to provide the tool to further research on which areas the different face APIs excel, and where they fall short.

The following sections are structured as follows: Chapter 2 will discuss the different tools and technologies that were used in building the MetaFace API suite. Chapter 3 will go into how the API was designed, documented, architected, and tested. Chapter 3 will also provide a demonstration of the requests and responses from the MetaFace APIs, as well as a front end that was constructed to demonstrate the API suite. Chapter 4 will discuss the results that were obtained as a result of this project, which were derived from one of the most popular public face datasets, UTKFace [54].

CHAPTER 2: REVIEW OF TOOLS AND TECHNOLOGIES

This chapter contains information about a broad spectrum of tools and technologies that are used in modern day software development. Because of the breadth of research, each topic is broadly covered with a high-level overview. Where applicable, each topic's application in the project is mentioned; however, the integration of the tools into a full solution stack will be detailed in chapter 4.

2.1 Programming Language Technologies

Thousands of programming languages make up today's computing world, but 5 languages make up about 50% of active code bases [6]. Any good computer scientist knows that each programming language has a space where it excels. There are plenty of languages, like Python, which serve as general-purpose languages that can be used in any application; however, this does not mean they should be used for everything. There are problem scopes where Python excels, and there are problem scopes where Java, C, and JavaScript excel. Before starting a new project, it is always important to think about which programming language will provide the performance, tools, and libraries to build the code base in a reliable and efficient manner [7].

2.1.1 JavaScript

The roots of JavaScript date back to the creation of the web browser. JavaScript was created in 1995 by Brendan Eich, who was on an extremely fast timeline to produce a programming language that enhanced web pages by animating components, validating user input, and more. The result was a language that was utilized by designers and web developers for many years and has grown to be one of the most popular programming

languages today, especially in the web application space [9]. JavaScript's original purpose was to dynamically script client-side web applications, but it is now used in applications for more than client-side scripting [1]. The rise of Node.js and the thousands of popular libraries that utilize Node.js have put JavaScript on the map as a server-side programming language. As the "size" of the Internet continues to grow, so does the popularity of JavaScript. GitHub, one of the most popular open-source and commercial code repositories, has measured that JavaScript is the most frequently committed code across all public and private projects, as shown in Figure 5 [5].

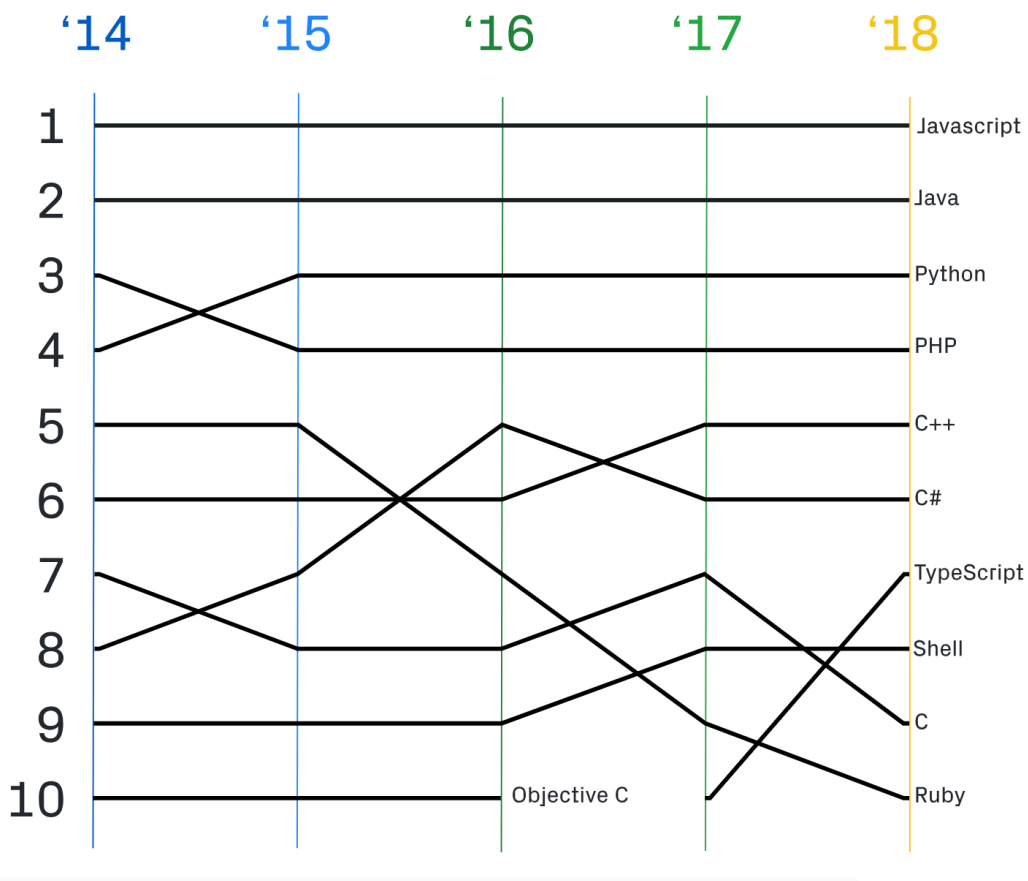


Figure 5 - Top Languages on GitHub Over Time [5]

JavaScript is often feared by developers, because it possesses such a diverse set of features. It is infamous for being error prone, because all features must be backwards compatible. If a feature of JavaScript were to become deprecated, millions of web pages that use old features would break [2]. Because of this, JavaScript can be found across the Internet in many different formats and shapes. Put another way, the same block of JavaScript code can be written many different ways, more so than other languages. An example of this is the developer choice between using callbacks, promises, and async/await. The new async/await feature should be used when developing new applications, but developers will find callbacks and promises spread all across tutorials and example applications when learning JavaScript, which makes it intimidating [3]. Although many developers stay away from JavaScript, it is extremely powerful in web and API development. Its' asynchronous behavior, which means no line of code can lock the program execution without explicitly telling it to, makes it complex, but it also makes it well suited to the event-driven request and response patterns [2]. JavaScript is the primary programming language that was used in this project, because of its popularity, event-driven design, and the vast array of libraries for building APIs and serverless applications that exist as open source.

2.1.2 ECMAScript

In the early days of JavaScript, source code was interpreted by something called a JavaScript engine. These engines were built into the browser and was how JavaScript was read for years. After the second browser war in 2009, JavaScript engines became much more powerful. Just as C and C++ can be compiled to machine code to be understood and

run by the processor of the computer, JavaScript engines used by browsers evolved to allow compilation of the source code to a lower level language that can be understood by a processor. This led to increased speed when loading web pages and applications, because a majority of the work could be diverted to the host computer, rather than done on the server side. From a high-level, a web browser retrieves an application from the Internet, which more than likely contains JavaScript, and compiles the JavaScript so that the application can utilize the processor of the host machine. There is a lot more that goes on behind the scenes, such as parsing for errors and caching, but it is not important for everyday JavaScript developers to know how each engine increases the response time of web events [1].

There are more than a few popular JavaScript engines. Mozilla Firefox is embedded with an engine called SpiderMonkey, Google Chrome is embedded with an engine called V8, and Chakra was developed by Microsoft for Internet Explorer and Edge. Each popular web browser has implemented their own JavaScript engine; however, they all compile the language in a way that keeps the behavior of the language true to the specification defined in ECMAScript. Each company continually improves the performance of their engine by optimizing compilation and interpretation times [8].

2.1.3 JSON, YAML, and XML

Different API and developer tools require/support different markup languages. Several parsers exist for porting between the popular markup languages like YAML, JSON, and XML; however, none of them are flawless. The only way to reliably code for

documentation generation and API definition is to write the documents in the native format for the tool.

JavaScript Object Notation (JSON) is an extremely popular data format. It utilizes collections of name and value pairs. It is based on a subset of the JavaScript programming language, so it is familiar and readable by programmers. It is also easy for programs to read, parse, and output JSON. There are two main data structures that the JSON format provides: objects and arrays. Objects are simply key value pairs, such as {“key”: “value”}. Arrays are ordered lists of values, such as [{“key1”: “val1”}, {“key2”: “val2”}]. JSON supports the same value types as most popular programming languages, such as strings, numbers, and Booleans, so it integrates seamlessly with programs, including APIs [17]. JSON is the primary request and response format of the Gladiator API (see Figure 6 for a sample of code).


```
{
    "serviceId": "4",
    "timestamp": "2019-04-10T18:30:26.659Z",
    "status": "complete",
    "result": {
        "type": "face",
        "boundingBox": {
            "x": 199,
            "width": 121,
            "y": 33,
            "height": 149
        },
        "attributes": {
            "frontal": true,
            "ageConfidence": 0.9454,
            "genderConfidence": 0.5796,
            "gender": "female",
            "age": 56
        }
    }
},
```


Figure 6 - Response from Gladiator in JSON

YAML stands for “YAML Ain’t Markup Language”. It is described as a human friendly data serialization standard for all programming languages. YAML relies on indentation for formatting. YAML is considered to be a strict superset of JSON, which means it can do everything that JSON can do, plus some [16]. YAML can be used for a variety of things, but its primary purpose in this project is API documentation (see Figure 7) and building CloudFormation templates, both of which will be discussed in later sections.

```yaml
paths:
  /:
    get:
      summary: Top-level resources and operations in this API
      description: >-
        Return links to the top-level resources and operations in this API.
          This
        API returns the following links:

        * **`gladiator:services`** : links to the
        collection of available face processing services.

      operationId: getApi
      responses:
        '200':
          description: OK
          content:
            application/json:
              example:
                id: services
                name: Face Processing Services
                apiVersion: 0.3.0
                _links:
                  'gladiator:services':
                    href: /services/services
      tags:
        - API
  /apiDoc:
    get:
      summary: Return API definition document
```

Figure 7 - API documentation in YAML

The Extensible Markup Language (XML) is an older text format. While many legacy systems still use XML, JSON has taken over as the standard for request and return data in APIs [17]. Many APIs still support XML as an input or output option by specifying a “Content-Type” field in the request header.

2.2 Node, NPM, and Node Packages

As previously mentioned, developers originally used JavaScript on the client-side and another language on the server-side, such as Ruby on Rails, PHP, Apache, or Spring. JavaScript was extended to a server-side language somewhat coincidentally. In 2009, Ryan Dahl realized JavaScript's problem of scaling to handle the many input/output processes concurrently and created Node.js, a server-side JavaScript framework that utilizes the V8 engine's event loop and low-level input/output API [11].

2.2.1 Node

The primary purpose of the Node project was, and still is, to allow JavaScript to become a server-side language; however, Node had to add some additional features to JavaScript before it could be used a server-side language. Because JavaScript was initially designed to be used by browsers, Node had to utilize an engine to compile JavaScript into a language that processors contained in web servers and computers can understand. Node uses the V8 engine to compile JavaScript into machine code, which can be read by processors that understand several different machine languages, including arm, x64, and mips. As with V8, Node is written in C++ because of its speed; however, Node also contains JavaScript code that is exported to use in Node projects. These JavaScript libraries are actually just wrappers for the C++ code so that JavaScript developers can use code that they understand without compiling C++.

Once Node had a way to translate JavaScript into a language that the web server could understand, other features had to be added to allow JavaScript to manage the server. Web servers often have to work with files, such as images and zip files, and

JavaScript contained no way of doing this, so Node added this functionality. Node also added the ability to deal with databases, since nearly all web servers deal with a database. Node also added the ability to communicate over the Internet via requests and responses in a standardized format, HTTP [8].

2.2.2 Node Package Manager (npm)

The Node Package Manager (npm) is the default package managing solution for Node.js. There are other package managers available, like Yarn from Facebook, but npm is the default. Over the last few years, npm has become the largest software registry. Npm has three components: a website, command line interface, and a registry. The website is where a developer can go to learn about a package that may help them with their software. The website also allows a user to setup a private repository so that only developers within an organization can use a package or set of packages. The command line interface runs from a terminal, which is what most developers use to install and manage versions of public or private packages. The registry is a public database of JavaScript packages, which also contains meta-information about each package, such as major and minor versions [12].

2.2.3 Express

Express is a Node.js framework that extends and simplifies the functionality of Node's web server. Doing things like sending a JPEG through an API would take at least 50 lines of code with Node's HTTP server. A group of developers grew tired with the complex interface of Node's HTTP server, and they built Express, which has grown to be one of the most popular node libraries with over 8,000,000 weekly downloads [21].

With Express, server functionality can be organized with middleware and routing. Middleware sounds like a complex term, but it's a fancy word for a library that handles a crosscutting feature in software, such as logging, parsing, or authorization. Express makes using and linking different middleware pieces as simple as a few lines of code. Express also comes with a router, which makes it easy to handle POST requests separately from GET requests [20].

Express is vital to this project, as it handles all of the request routing, parsing of requests and responses, and delivery of responses. Parsing was done through a library provided by Express called Body-Parser [34]. Body-Parser is used as middleware and handles parsing of JSON objects and allows URLs to be encoded in responses. Using Express, along with Node, made it extremely easy to organize the code into modules and handle business logic within each request endpoint.

2.2.4 Watson and AWS SDKs

Many larger companies provide software development kits (SDKs) to work with their APIs. Common programming languages that companies provide SDKs for are Java, Python, Ruby, and JavaScript. These SDKs usually provide the same features that the API provides, but it allows developers to work with APIs in their native language, without needing to handle retry logic and error handling in HTTP. IBM Watson and AWS Rekognition both provide SDKs in Node, called Watson-developer-cloud [32] and aws-sdk [33], and they were utilized in this project to simplify working with IBM's and AWS' APIs.

2.3 Developer Tools

This section focuses on the Node libraries that were used to format, test, and run the code that was written in JavaScript, powered by Node, and supported by Express. The project could still work and function without these libraries, but they each serve a unique purpose that makes the API more reliable, scalable, and maintainable. How this project handles qualities such as reliability, scalability, maintainability, cost optimization, and security will be discussed in chapter 3.

2.3.1 Gulp

Gulp is mostly used as a build tool for front end and back end systems. It does very simple things, but it does them efficiently. For example, it can require that linting, formatting, and testing be run before a project is compiled and deployed. This may sound trivial, but the alternatives are doing it manually or writing complex npm scripts. Gulp allows developers to setup fast pipelines for building and deploying projects. The developer is in control of which tasks are performed asynchronously and synchronously. The dependencies between these mixed modality tasks can be defined by the build process for consistent behavior of each build [23]. Gulp is actually not used in the build and deployment of the API, as there exists another Node library for deploying serverless APIs; however, Gulp is used to generate and deploy the web pages for the API documentation.

2.3.2 Formatting with Prettier

Anyone that has been coding for more than a few months will tell you that formatting matters. There are plenty of formatting standards that developers agree on,

such as how a code block following an “if” statement should be indented. Although, there is less agreement on where a bracket that encloses the body of that if statement should go. Some say that it should go on the same line as the “if”, while others say that it should go on a new line, just below the “if”. A lot of organizations setup formatting standards and expect all development teams to follow them. However, independent developers develop their own formatting standards based on their style.

Enter Prettier, which has become the most popular code formatter out there with 4,000,000 weekly downloads in Node alone [21]. Prettier allows organizations or developers to define their own preferences for each popular formatting rule in a markup language like JSON, such as {“singleQuote”: true}, which states that a string should be enclosed by single quotes rather than double quotes. Furthermore, prettier will actually format the entire code base whenever it’s saved, if configured. Prettier can even format JSON and YAML [24]. Those adopting this code base should follow the style of the code base, defined in the .prettierrc file, for simplicity and readability throughout the project.

2.3.3 Linting

A linter is a tool that flags code errors, bugs, and style errors, usually before code is compiled or built. Prettier negates some of these flags by ensuring they are fixed before the linter is even run, but a good linter can pick up things that Prettier doesn’t fix, such as prohibiting console.log() functions in production code. Linters will also pick up things like variable names that are declared multiple times. ESLint is a very popular linter for modern JavaScript, and it is used by major companies such as Facebook, Airbnb, PayPal, Atlassian, Microsoft, and Netflix [25]. Airbnb has publicized their formatting standards

using ESLint so that other projects can extend their rules [29].

```
{
  "extends": ["airbnb", "prettier", "plugin:jasmine/recommended"],
  "plugins": ["prettier", "jasmine"],
  "env": {
    "node": true,
    "es6": true,
    "browser": true,
    "jasmine": true
  },
  "rules": {
    "func-names": 0,
    "max-len": 0,
    "no-underscore-dangle": 0,
    "no-use-before-define": 0,
    "no-unused-vars": 1,
    "prettier/prettier": ["error"],
    "no-else-return": 0,
    "consistent-return": 1,
    "jasmine/no-focused-tests": 0
  }
}
```

Figure 8 - ESLint configuration document

ESLint was used in this project, and the Airbnb format was extended. The rules that were changed from Airbnb's standards are shown in Figure 8 above. A value of '0' means that no warning or error is thrown for a particular row, while a value of '1' means that a warning should be displayed. The rest of Airbnb's linting errors were kept. The Airbnb style guide can be found at https://github.com/airbnb/javascript. An example output from ESLint can be seen in Figure 9 below. From the output below, you can see warnings present. This helps to know where console.logs() are at all times, so a developer doesn't accidentally leave a data leaking debug statement in their code.

```
MacBook-Pro:gladiator Kevin$ npm run lint

> gladiator@1.0.0 lint /Users/Kevin/Programming/CapstoneProject/gladiator
> eslint ./src/**/*.js


/Users/Kevin/Programming/CapstoneProject/gladiator/src/function/getResults.js
  31:7  warning  Unexpected console statement  no-console

/Users/Kevin/Programming/CapstoneProject/gladiator/src/function/gladiator.js
  16:35  warning  Expected to return a value at the end of arrow function  consistent-return
  39:5   warning  Unexpected console statement                             no-console
  43:9   warning  Unexpected console statement                             no-console
  46:9   warning  Unexpected console statement                             no-console

✖ 5 problems (0 errors, 5 warnings)
```

Figure 9 - Linter console output

2.3.4 Unit & Integration Testing

With the increase in companies that are pushing test-driven development and behavior-driven development, it is no surprise that there are several popular libraries for testing the behavior of APIs. There are some which perform very specific tasks, such as the assertion library, Chai [30]. Others try to stay general and provide classes and methods for all areas of testing. While there are hundreds of testing libraries out there, only the ones used in this project will be discussed.

Jasmine is a unit testing framework that that is robust enough to be the only unit testing or integration testing required for a project. Unit testing is the act of testing a single module or function of code, while integration testing is testing how the modules interact with one another (Figure 10 shows an example unit test in Jasmine). Jasmine provides asserts, mocks, spies, and test definitions. In this project, Jasmine is used primarily for asserts, spies, and test definitions [22].

- Asserts compare two variables and pass/fail if they do or do not meet a certain condition.

- Spies are entities that monitor variables and functions and ensures that they go through certain state changes, such as being altered or called a certain number of times.
- Test definitions are simply a way of organizing tests by module and code path. An example of this test definition is shown below, where ‘describe’ is used to separate different modules and functions, and ‘it’ is used to define individual tests.

```
describe('getService.js', () => {
  afterEach(async () => {
    await AWS.restore();
  });

  it('should get a service if all parameters are given and dynamo returns an item', async done => {
    await AWS.mock('DynamoDB.DocumentClient', 'get', (params, callback) => {
      callback(null, getMockedServiceDbResponse());
    });
    try {
      await request(getService)
        .get('/services/1')
        .set('Accept', 'application/json')
        .set('Content-Type', 'application/json')
        .expect(200, getMockedServiceDbResponse().Item);
    } catch (err) {
      fail(`expected test to pass and error was thrown: ${err}`);
    }
    done();
  });

  it('should return 404 error if id does not exist in db', async done => {      You, 9 days ago • ad
    await AWS.mock('DynamoDB.DocumentClient', 'get', (params, callback) => {
      callback(null, {});
    });
    try {
      await request(getService)
        .get('/services/1')
        .set('Accept', 'application/json')
        .set('Content-Type', 'application/json')
        .expect(404);
    } catch (err) {
      fail(`expected test to pass and error was thrown: ${err}`);
    }
    done();
  });
```

Figure 10 - Unit test example

Mocking is huge part of testing. There is no need to actually call out to the production database whenever running tests. Some developers will actually setup a test database and connect to that; however, that introduces some overhead. If the records or entities in a database get into a bad state, you'll spend more time cleaning up and restoring the database, making this process inefficient. If you know how the database should respond, then you can mock that response and pass it into your module. A mock

simply means an example response from another piece of your code or third-party library or service that is not currently being tested. In this project, the aws-sdk-mock library [26] was used to mock the response from AWS resources (covered in next major section). The main AWS resource that is mocked is the database, which is called DynamoDB [13]. Another mocking library that is used is supertest, which is a library used to mock HTTP requests. Rather than actually spinning up a localdev server for Express to use for GET and POST requests, supertest mocks all of the server behavior so that developers can focus on writing the tests [27].

2.3.5 Code Coverage

Code coverage is an extension of testing. It provides a percentage of the code that was covered by the suite of unit or integration tests. Most code coverage libraries allow you to view the percentage of code covered by branch, line, or function. Nyc is one of those libraries. It integrates with Jasmine and other popular testing frameworks. It even integrates with different test reporters to give different views of the coverage results, and one of these even delivers a graphical heatmap of which code is covered and uncovered [28]. Two examples of these coverage results are shown in Figures 11 and 12, using different reporters.

```
-----------------------|---------|----------|---------|---------|-------------------
File                   | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------|---------|----------|---------|---------|-------------------
All files              |   90.74 |    81.58 |   96.15 |   90.61 |
 function              |   86.62 |       75 |   94.74 |   86.43 |
  getApiDoc.js         |     100 |      100 |     100 |     100 |
  getResults.js        |   81.82 |    72.73 |   93.33 |   81.63 | ... 85,188,192,194
  getRoot.js           |     100 |      100 |     100 |     100 |
  gladiator.js         |   96.15 |     87.5 |     100 |   96.15 | 26
 function/services     |   98.65 |    95.83 |     100 |   98.63 |
  createService.js     |   96.67 |    93.75 |     100 |   96.67 | 24
  getService.js        |     100 |      100 |     100 |     100 |
  getServices.js       |     100 |      100 |     100 |     100 |
-----------------------|---------|----------|---------|---------|-------------------

=============================== Coverage summary ===============================
Statements   : 90.74% ( 196/216 )
Branches     : 81.58% ( 62/76 )
Functions    : 96.15% ( 25/26 )
Lines        : 90.61% ( 193/213 )
================================================================================
```

Figure 11 - Code coverage console output

All files / function/services getService.js

100% Statements 23/23 **100%** Branches 4/4 **100%** Functions 2/2 **100%** Lines 23/23

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1  1x  const serverless = require('serverless-http');
2  1x  const express = require('express');
3  1x  const bodyParser = require('body-parser');
4  1x  const AWS = require('aws-sdk');
5
6  1x  const app = express();
7
8  1x  const { NotFound } = require('../../error/notFound');
9
10 1x  app.use(
11       bodyParser.urlencoded({
12         extended: true,
13       })
14     );
15 1x  app.use(bodyParser.json());
16
17 1x  app.get('/services/:serviceId', (req, res) => {
18 3x    const { SERVICES_TABLE } = process.env;
19 3x    const dynamoDb = new AWS.DynamoDB.DocumentClient();
20
21 3x    const params = {
22         TableName: SERVICES_TABLE,
23         Key: {
24           serviceId: req.params.serviceId,
25         },
26         AttributesToGet: ['serviceId', 'serviceName', 'endpointUri'],
27       };
28
29 3x    dynamoDb.get(params, (error, result) => {
30 3x      if (error) {
31 1x        console.log(error);
32 1x        return res.status(500).json({ error: 'Could not get service' });
```

Figure 12 - UI for code coverage heatmap

2.4 Amazon Web Services (AWS)

Cloud computing is the on-demand delivery of information technology resources and services through the Internet where you only pay for the resources that you use. For the past 10 years, many major corporations and small businesses have been flocking to a cloud provider because of the cost savings, reliability, scalability, and ease of use that they provide for resources like storage, server space, machine learning, security, and more. Businesses no longer have to pay to upgrade server power or buy new hard drives every few months or years, as they can piggyback on a large-scale cloud provider like Amazon, Google, Microsoft, or IBM.

According to the RightScale State of the Cloud survey of 2019 (Figure 13), which surveys 456 enterprises and 330 small businesses, Amazon is the most popular cloud provider of 2019 [31]. Amazon had a shopping platform that was used by millions of people, and they realized that the services they had built to support that platform could be spun off into another product. Thus, Amazon Web Services (AWS) was born. Amazon quickly generalized all of their services so that any business could plug into them, and their services are all independent such that you can use one of their components with a company's existing infrastructure or use their entire platform for an application. AWS provides hundreds of services at the time of writing, and they continue to add more every year [14]. Because of the widespread use of AWS across millions of customers, AWS was used for the infrastructure for this project. Although Amazon provides many services for a range of applications, only the services that were used in this project will be discussed.

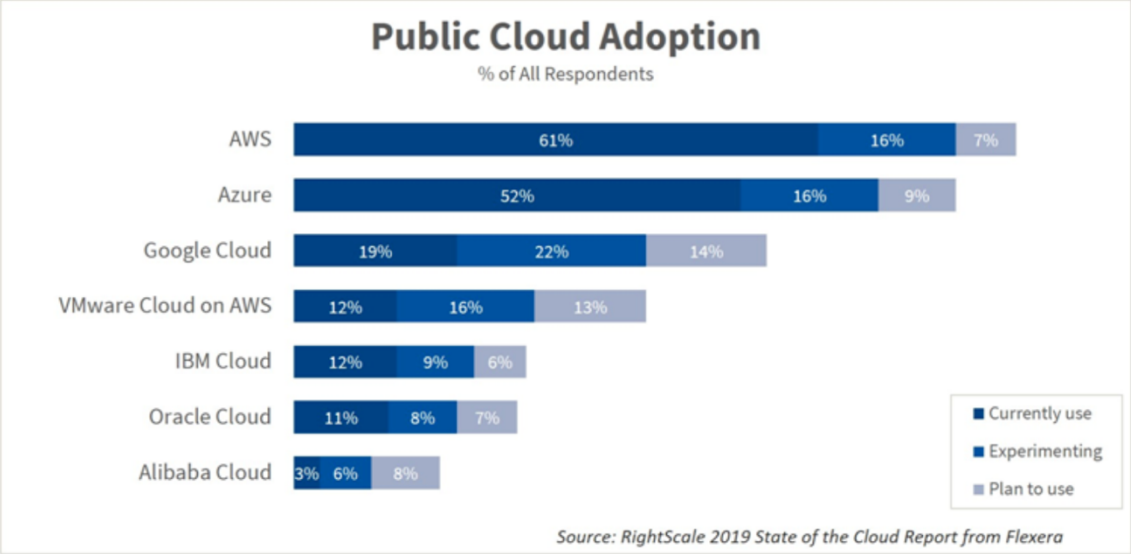


Figure 13 - State of the Cloud Survey 2019 results for most popular cloud provider

2.4.1 Simple Storage Service (S3)

Amazon's Simple Storage Service (S3) provides developers and IT teams with secure, durable, and highly-scalable object storage. The data is automatically spread across multiple devices and facilities. Object storage means that the user can upload files, from 0 bytes to 5TB, to the cloud. The files are stored in an entity called a bucket, which can have folders as well. S3 names must be globally unique, because they are all given a URL for ease of access. An example URL is https://s3-eu-west-1.amazonaws.com/exampleBucket [14].

S3 is built for 99.99% availability, which means occurrences where data cannot be accessed should be extremely rare. S3 is also built for 99.999999999% (11 9s) durability, which means data should almost never be lost [13]. S3 provides different tiers for different levels of access so that the user can save money on infrequently accessed data or archives. S3 supports many extra features that the user must turn on, such as

lifecycle management, versioning, encryption, and access control rules and policies. By default, all newly created buckets are private; however, you can allow access to an S3 URL by configuring the access control list and bucket policies [15]. In this project, S3 stores all of the images that are sent to the API via a POST request.

2.4.2 Identity and Access Management (IAM)

The Identity and Access Management (IAM) service allows you to create sub users for a particular AWS account. This is important for large organizations that need to setup users or groups of users for each developer, sales representative, software tester, or any other role in an organization [14]. What is also useful about IAM is the concept of roles, which allow you to give a certain set of permissions, called a policy, to any person or other AWS resource. For instance, you can give a lambda function a role that allows it to write and read to a DynamoDB table or S3 bucket. AWS provides thousands of pre-configured roles that are commonly used, but users can also create custom roles for their particular use case [15]. In this project, custom IAM policies allow any interactions between necessary resources and reject all others. An example of a policy document is shown in Figure 14.

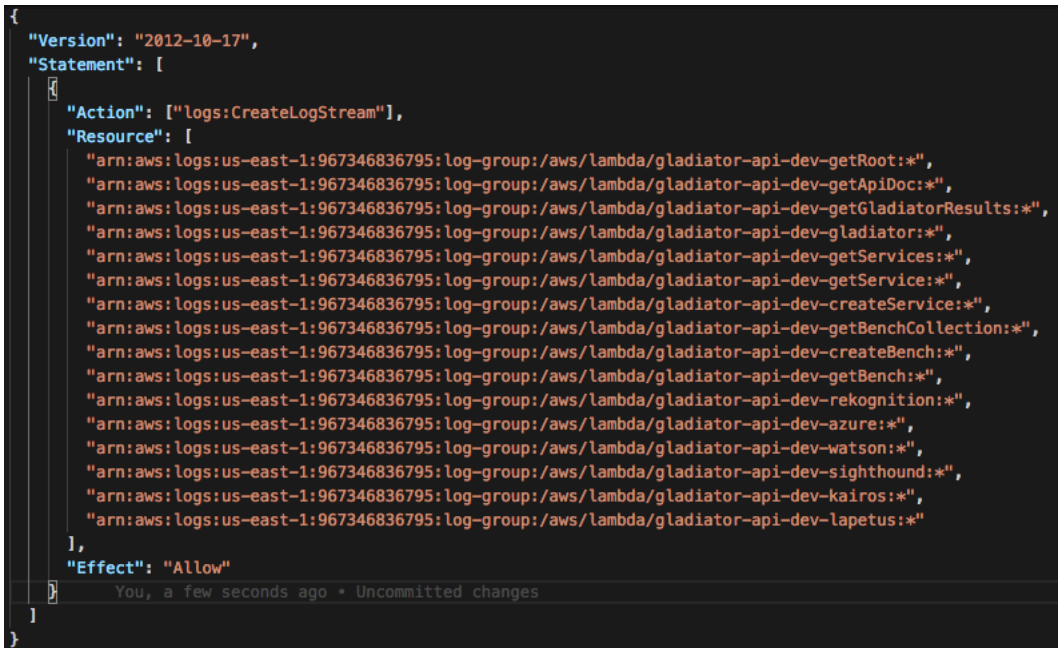


Figure 14 - An example policy document that allows API endpoints to write to CloudWatch logs.

2.4.3 Lambda

Lambda is becoming one of the most popular services offered by AWS. With lambda, developers can upload their code and run it without having to provision, manage, or upgrade servers. AWS provisions and manages the servers behind the scenes, and developers are freed from the worries of operating systems and scaling. Lambda is often used in response to an event, such as an API Gateway call or a CloudWatch event. Lambda events can also trigger other lambda events, so a single lambda function can invoke an infinite number of lambda functions. Lambda scales out automatically, but not up. If an application is called 1 million times, then 1 million invocations of that lambda function will be triggered; however, if one Lambda instance needs more resources, Lambda will not increase the memory allocation automatically.

Lambda provides runtime environments in many popular programming

languages, including C#, Go, Java, Node.js, and Python. Lambda can be triggered by API Gateway, Alexa, CloudFront, CloudWatch, DynamoDB, Kinesis, S3, and SNS [13]. At the time of writing, Lambda only costs $0.20 per 1 million requests plus whatever compute time each lambda executes, which is capped at 15 minutes. Services like Lambda have created a new architecture design known as serverless, which is highly scalable and low cost [15]. Lambda and the idea of serverless technology, which will be discussed in chapter 3, is utilized heavily in this project.

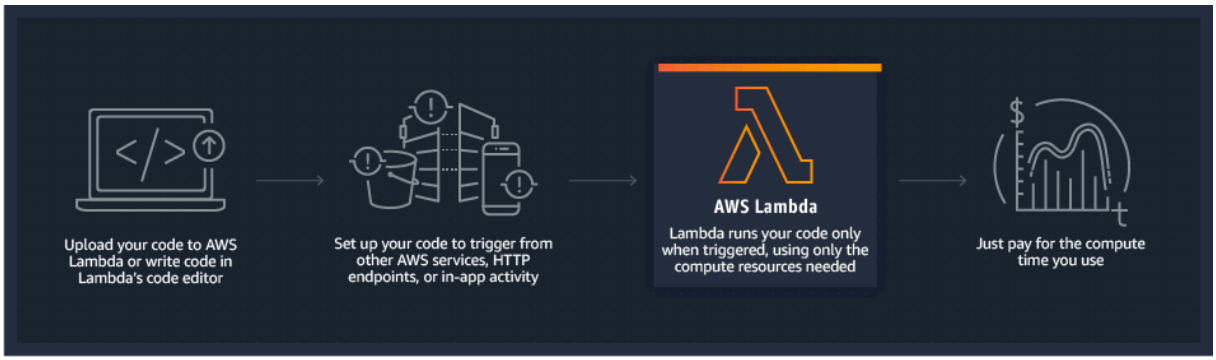


Figure 15 - AWS graphic describing Lambda [13]

2.4.4 API Gateway

Amazon's API Gateway is a service that provides the functionality to create and publish APIs at any scale. API Gateway can act as a front door to data and functionality from backend services, whether they're hosted on AWS or on an existing infrastructure. API Gateway integrates seamlessly with Lambda, so it is perfect for designing and building serverless APIs. Another reason that API Gateway pairs well with serverless APIs is that it can handle the processing of up to hundreds of thousands of concurrent API calls. It controls traffic management, access control, and even version management [14].

2.4.5 DynamoDB

DynamoDB is the NoSQL database service offered by Amazon. It provides consistent, single-digit millisecond latency at any scale. The data is automatically replicated across 3 geographically distinct data centers, so it can be used in compliance with regulations from the consumer to the finance sector. DynamoDB uses key value pairs as a storage mechanism, so it fits well with the JSON data format that is preferred by most modern APIs [13].

2.4.6 Route53

Route53 is a domain name system, which provides a cost effective and reliable method of routing end users to web applications by translating human readable names, such as www.website.com, to IP addresses, such as 192.0.0.1 [13]. Route53 can be used in very complex ways, but this project only utilizes it for its basic purpose of routing URLs to the IP of S3 buckets with hosted websites.

2.4.7 Simple Notification Service (SNS)

The Simple Notification Service (SNS) delivers messages between AWS resources or to end users. SNS excels in situations that require high-throughput, push-based notifications. With SNS, publisher systems can fan out messages to infinitely many subscribers for parallel processing, including Lambda functions. Subscribers do not need to keep polling for updates from SNS [15]. SNS is the service that drives concurrent request and response parsing from the third-party face processing APIs.

2.4.8 CloudWatch

CloudWatch is a service that allows monitoring of other AWS resources for usage

based on different metrics. With CloudWatch, a user can create different dashboards to show you metrics of interest for a particular application. CloudWatch also allows users to configure alarms, which send alerts to the administrator when a certain threshold is passed. These alerts can be sent via text or email. CloudWatch also provides events, which allow an application to respond to state changes in resources without admin interference. A very important part of CloudWatch is the logs (see Figure 16 for an example). This is where the application can store information about exceptions, errors, and warnings [15]. Figure 17 shows an example of a custom graph that shows the number of error responses that were returned between April 14, 2019 and April 27, 2019.

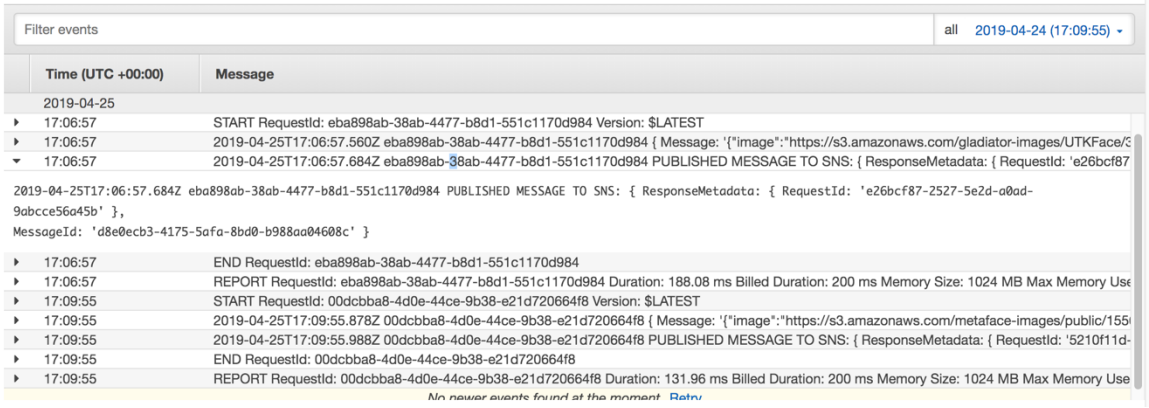


Figure 16 - CloudWatch Log from Gladiator Lambda

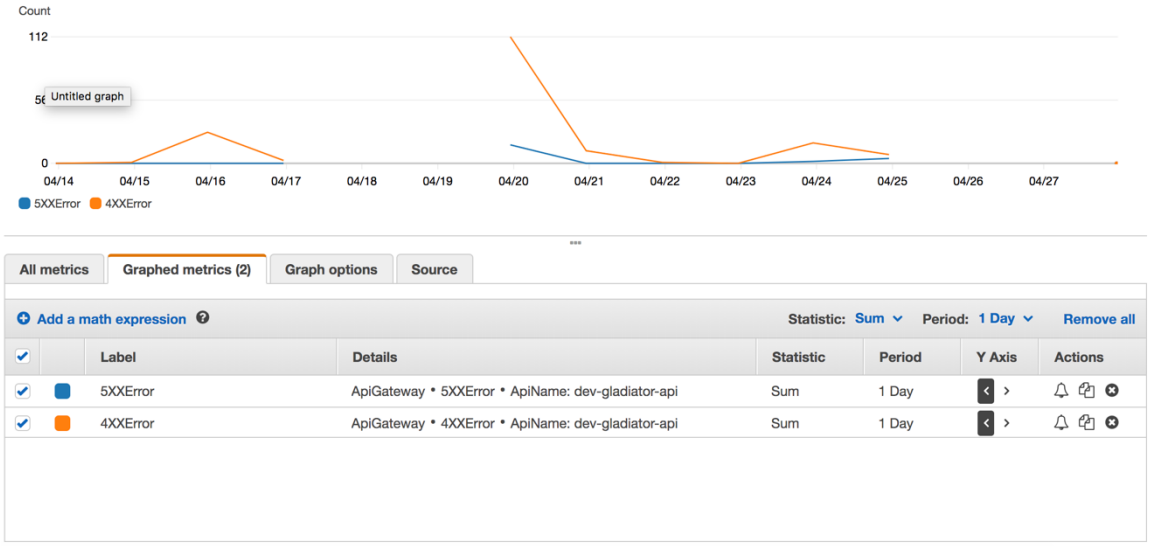


Figure 17 - CloudWatch Metrics showing number of 4xx and 5xx API error responses for 2-week period

2.4.9 CloudFormation

AWS CloudFormation provides a common language for you to describe and provision all the infrastructure resources in your cloud environment. Figure 18 shows the workflow for utilization of CloudFormation. Designing an AWS architecture in CloudFormation templates allows developers to experiment with new patterns with ease. The template can be deployed to a production environment, then used in numerous sandbox environments for development and testing. CloudFormation templates also provide the benefit of resource recovery if an environment is broken or compromised [14].

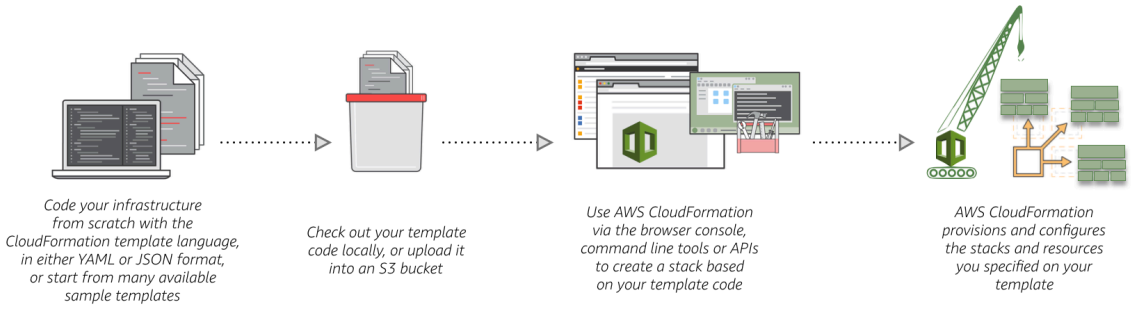


Figure 18 - AWS graphic that shows workflow of CloudFormation [13]

2.4.10 Cognito

Cognito provides two features: the user pool and federated identities, also known as the identity pool. The two sub services are often used in coordination with one another. The user pool provides a way for applications to handle user registration, authentication, and account recovery via “forgot your password” or “forgot your username”. The identity pool provides a way to authorize users and give them access to a specific subset of AWS resources in an application (see Figure 19 for a process flow). For example, if a user needs access to S3, identity pools provide a method of giving access to only the buckets, folders, or files that they “own” in S3. The identity pool has an independent concept of an identity [13]. This identity could represent a user in a Cognito user pool, or even a user from Facebook or Google [19].

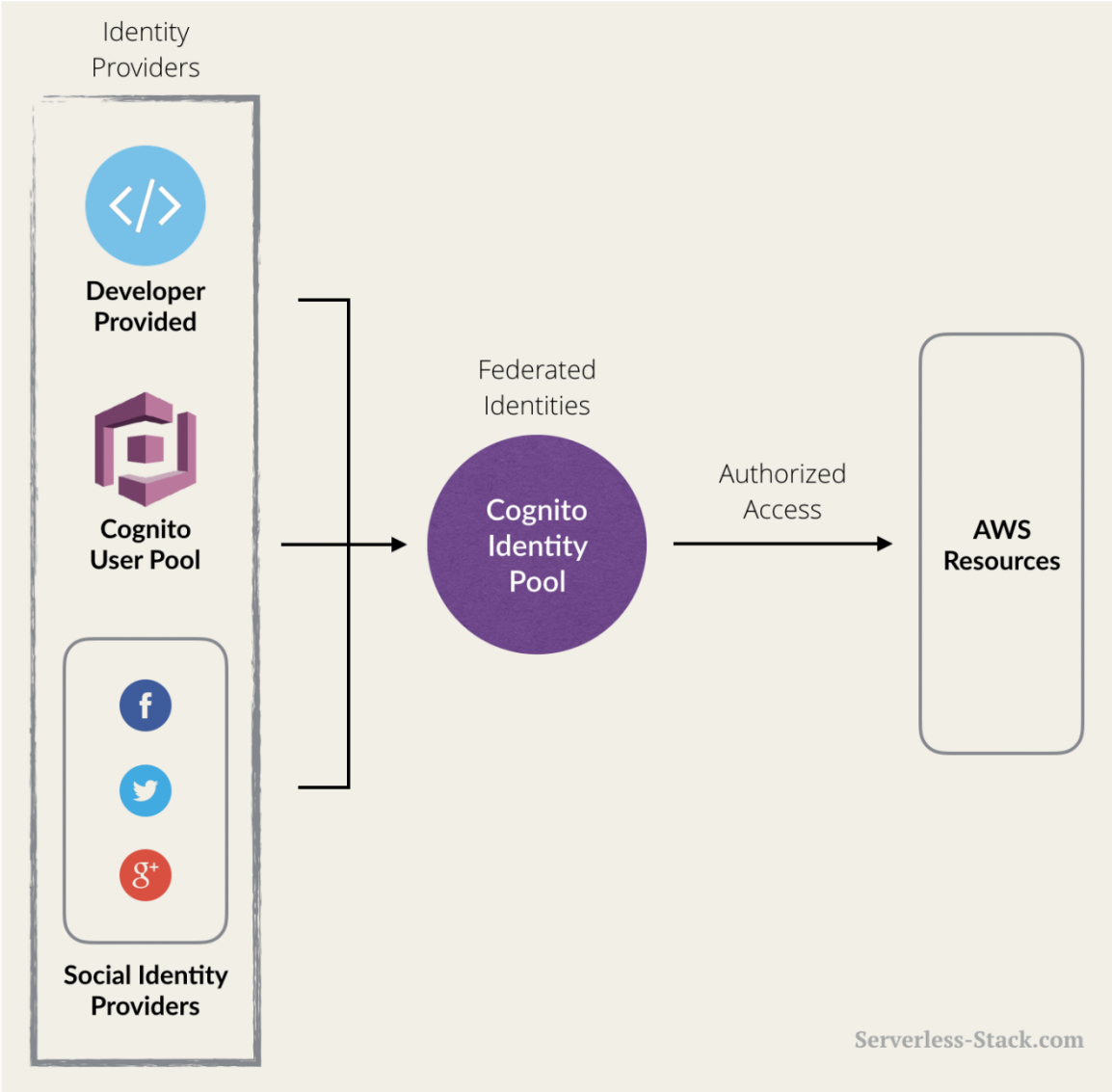


Figure 19 - Cognito functionality from Serverless-stack.com [19]

2.4.11 Amplify

Serverless stacks and APIs are so popular in AWS that Amazon implemented their own open source framework to build serverless mobile and web applications (Figure 20). It provides "an opinionated set of libraries, UI components, and a command line

interface to build an app's backend and integrate it with your iOS, Android, Web, and React Native apps" [19]. Although its primary use is for mobile applications, it can be used for web applications in Vue, React, and Angular. It can also be used to make virtual reality apps [13].

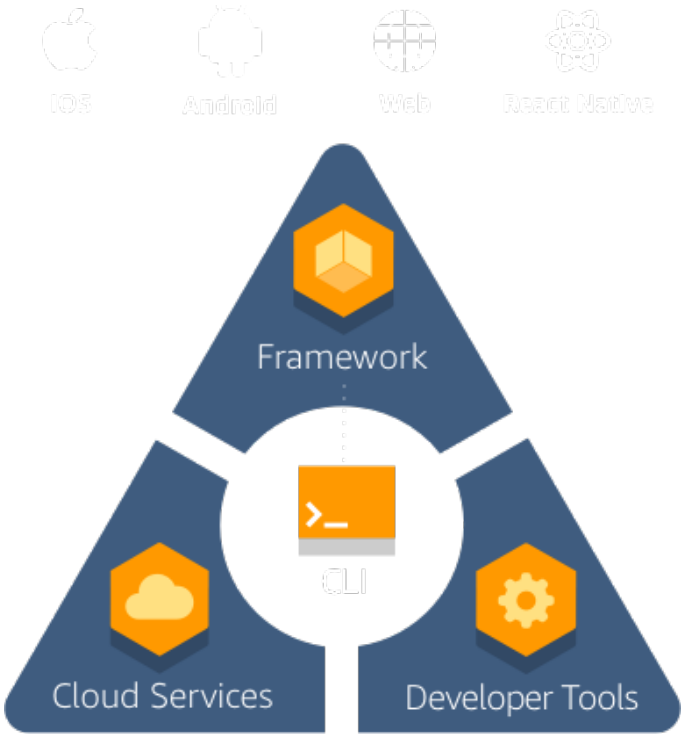


Figure 20 - AWS graphic presenting features of Amplify [13]

2.5 Serverless Framework

The Serverless Framework is a tool for building and deploying serverless applications. The serverless architecture pattern will be discussed in chapter 3, but knowledge of the pattern is not needed at this time. Essentially, the Serverless Framework lets you define your APIs and resources, and it will handle creation of the CloudFormation templates and deploying them to AWS, which then creates all of the

resources for a serverless REST API [37]. In this project, the entire serverless application infrastructure is defined within a single YAML file. The express apps are consumed and deployed into a Lambda function by a Serverless library called serverless-http [38]. Serverless supports many programming languages, including Node, and it also supports multiple cloud providers, including AWS. Serverless also handles the streaming of logs for debugging and troubleshooting [37]. Serverless will be discussed further throughout the paper.

```yaml
functions:
  getRoot:
    handler: src/function/getRoot.handler
    package:
      include:
        - src/function/getRoot.js
    events:
      - http:
          path: /
          method: GET
          cors: true

  getApiDoc:
    handler: src/function/getApiDoc.handler
    package:
      include:
        - src/openapi/index.yaml
        - src/function/getApiDoc.js
    events:
      - http:
          path: apiDoc
          method: GET
          cors: true

  getGladiatorResults:
    handler: src/function/getResults.handler
    package:
      include:
        - src/function/getResults.js
    events:
      - http:
          path: gladiator/{resultId}
          method: GET
          cors: true
          private: true
    timeout: 30
```

Figure 21 - Excerpt from serverless definition file

CHAPTER 3: METHODOLOGY

This chapter describes the ideas, designs, patterns, and best practices that are used in the project. These methods are standardized in several popular APIs for design, implementation, and documentation.

3.1 API Design & Implementation

There are many aspects of API design that differ from traditional software development. One of the main differences between developing web, mobile, or desktop applications and APIs is that the major consumer of APIs is other developers. When offering an API as a product, you have to think from the perspective of the developers that will consume the APIs in their own applications. For developers to feel comfortable using an API, it must be efficient, well-designed, intuitive, easy to integrate, and heavily documented. Several patterns and tools exist to help design and create standardized APIs [35].

3.1.1 REST & HATEOAS

Architectural styles are a predefined solution structure that allows an API to be built more swiftly and consistently than without a plan. The most popular style is REST, which stands for Representational State Transfer. It defines a set of constraints and agreements that are designed to use the HTTP-based infrastructure and protocols. The constraints that make up the REST architecture are use of HTTP protocols, design of resources instead of operations, stateless communication between the server and client, loose coupling and independence of requests, and use of HTTP status codes and media types.

REST uses resources as its central concept. A resource is an abstract data model,

which can contain sub resources. Resources and sub resources can be found using their unique URI, which stands for Uniform Resource Identifier. A resource can be serialized to various representations, which can be JSON or XML. Other representations can be used, but JSON and XML are the standard in modern APIs. It is important to note that the information contained in both representations should be the same, and many XML to JSON and JSON to XML parsers are available to translate from one representation to another. HTTP methods are used to obtain, create, update, and delete resources, and not all methods may be supported by every resource.

REST APIs mostly deal with CRUD operations, which mean operations that create, read, update, and delete. Reads can be mapped to the HTTP method GET, creation can be mapped to POST, update can be mapped to PUT or PATCH, and delete can be mapped to DELETE. Each HTTP method serves a specific purpose and is idempotent, meaning methods can be executed multiple times without altering the end result. This supports a stateless architecture because the server does not need to remember any information about resources if all of the necessary application information is passed with every HTTP request. In the stateless architecture, only application state needs to be maintained by the application. Application state keeps track of the interaction in a single instance of the client. Application state should not be shared between multiple client applications. APIs should not maintain application state, as there may be multiple types of applications making requests to them at any time. Each API call should be independent of the one before it, and so it must contain all of the necessary information to perform the action. The bottom line is that application state must be maintained by the consumer, who

is the developer of the application. Application state is different from resource state, which stores information about the state of business objects. Resource state is shared by all applications and their instances. Resource state is stored on the server, usually in a database. The HTTP methods POST, PUT, PATCH, and DELETE are used to change the resource states through the API.

The main advantage of REST is its scalability. REST is easy to scale because it does not have to maintain application state and each request is independent. Any time the server starts to fall behind with an increased number of consumers, more servers can easily be added behind a load balancer. Another advantage is the use of caching under the HTTP infrastructure. REST also supports the use of several data representations, so that the developer can get the data in the format that they are comfortable with.

HATEOAS is an extension of REST that allows the client to explore the API without any knowledge about the relationships between resources. Each resource contains hyperlinks to other related resources. The client only needs to make a call to the root resource, then it can follow links to other operations on the current resource or operations on other resources. With HATEOAS, the number of requests that the client has to remember are minimized. The HATEOAS style can be modeled as a state machine, because you can track the links that can be followed beginning with the root resource. For instance, say you make a GET request to a bank account. The bank account resource might contain links to make a transfer to another bank account, close the bank account, or deposit money into the account, depending on the use case of the API. One major advantage of HATEOAS is flexibility, because changes can be introduced without

breaking any clients as long as the links remain the same. The client also does not need to learn as much about the API to integrate it into their application [35].

3.1.2 Error Handling

APIs use the response status to tell whether a request was successful or encountered an error. HTTP has defined status codes that tell which success or error status code should be returned for each scenario. For a complete list of status codes, see https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. Generally, 1xx codes are informational, 2xx codes are success codes, 3xx messages are redirection codes, 4xx messages represent client errors, and 5xx messages represent server errors. For example, a status code of 200 means that a resource was retrieved, status codes of 400 mean that a request was not formed correctly, and status codes of 404 mean that a resource could not be found [36]. This project only utilizes the status codes defined above, as well as the 500-status code for database errors. Some endpoints also contain an extra status property. This status property is set to failed whenever a particular third-party API returns an error, such as no faces found in an image, image size too large, or invalid image URL.

3.2 API Documentation

The first point of contact between a developer and the API itself is the documentation, so a developer will likely choose another provider if they cannot understand the documentation. There are a few popular API description languages that are used for documenting APIs. Contrary to programming languages, API description languages tell the “what” rather than the “how”. Developers can look at API documentation and know what endpoints are provided, what each one does, how they can

pass information to the API, and what the response should look like. API descriptions are primarily for consumers of the API, and they should not document things like backend connections. These will have to be documented separately for internal developers of the API. Although the API description language's primary purpose is providing human readable API documentation, they can also be used to generate code skeletons for implementation, documentation web pages, and tests [35].

3.2.1 Swagger and OpenAPI

OpenAPI is a specification for defining and describing REST APIs in a design specification document. Some of the things that you can document using an OpenAPI file include available endpoints and paths, the operations supported for each endpoint, the parameters (path, query, body), authentication methods, and contact information. OpenAPI specification documents can be written in JSON or YAML format. OpenAPI is defined by an organization, similar to how ECMAScript defines JavaScript, and determines how REST APIs should be designed and implemented. For an example of OpenAPI, see Figure 22.

```yaml
paths:
  /gladiator:
    post:
      summary: Post an image, get results
      operationId: postGladiator
      requestBody:
        $ref: '#/components/requestBodies/gladiatorInput'
      parameters:
        - $ref: '#/components/parameters/includeParam'
        - $ref: '#/components/parameters/excludeParam'
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/gladiatorOutput'
      tags:
        - Gladiator

components:
  schemas:
    gladiatorInput:
      title: Gladiator Input
      description: >-
        The information that must be provided to get results from all of the face
        processing services.
      type: object
      allOf:
        - $ref: '#/components/schemas/gladiatorFields'
      properties:
        apiKey:
          type: string
          description: The API key that is given to the user after registration
          example: abcdef12345
```

Figure 22 - API documentation using OpenAPI 3.0

Swagger is toolset provided to design and implement APIs according to the OpenAPI specification. It provides an editor for editing the YAML or JSON file, as well as a viewer for displaying the specification in a clean and readable webpage. Swagger also provides a code generation service to generate server stubs from the specification document, which save developers hours of writing code. Lastly, Swagger provides a way to host the design documents on a website.

3.2.2 Project Endpoints and Methods

In this project, 4 endpoints are proposed: root, services, gladiator, and bench. These endpoints were designed such that each is independent and serves a unique purpose. Each endpoint's documentation pages can be found in Appendix A.

Three endpoints are provided by MetaFace: Services, Gladiator, and Bench. The 'Services' endpoint keeps up with the API keys and endpoints that is used to get results from each third-party API. The 'Gladiator' endpoint controls the submitting of images and retrieval of results from the services stored in the 'Services' endpoint. The 'Bench' endpoint handles the retrieval of benchmark results for each dataset that is run by an admin. Perhaps in the future, users can submit dataset results to the 'Bench' endpoint to help with research. The root endpoint just returns links to the supported endpoints. A 'Users' endpoint was considered but deemed unnecessary since API keys are created using AWS API Gateway and users are created and authenticated using AWS Cognito.

These endpoints meet the standards set by the REST and HATEOAS design styles because each endpoint controls an individual and independent resource. Each POST, PATCH, or DELETE request can only create or update a single instance of a resource, and the GET method retrieves a single instance or collection of a resource. Furthermore, an application does not need to maintain state information about the API. All of the resource state information is stored at the API level. The API design also follows the HATEOAS design because it contains links between the endpoints where applicable.

3.2.3 Documentation Generation

There are those who question the writing of design specification documents, especially in small scale APIs; however, there are many benefits even for an API with one or two endpoints. If the API is going to be used by anyone other than the developer that designed and implemented the API, then it needs to be intuitive and easy to use. Even though the OpenAPI document in YAML or JSON is easy to read, it takes a long

time to comprehend what each endpoint is doing and how everything relates. The Swagger UI tool provides a simple web design that an end user can read and understand what each endpoint does, but the tools don't end there. There exist several libraries that will generate entire documentation web pages from OpenAPI specifications. ReDoc is the open-source tool that was used in this project, and the result is hosted at http://www.docs.gladiatorapi.com. A few screenshots from the generated web page are included in Appendix B. It is important to note that no manipulation of the html, css, or js files was required, and it generates a web page that can be viewed on mobile devices as well (shown in Appendix B). The build and deployment process are completely automated. This means that if changes are made to the specification, the documentation and web page can be regenerated and deployed within seconds!

3.3 Architecture

The Serverless Framework was created to simplify the process of creating serverless applications. Serverless applications have risen in popularity in recent years, because they are cost efficient, scalable, and easy to maintain, but what does serverless mean? [39]

3.3.1 Serverless Concept

The Serverless Architecture pattern, which has also been called function as a service (FaaS), is a design pattern where applications are hosted by cloud providers, which eliminates the need for server and hardware management by developers. Instead of an entire application being hosted on servers, each function is individually invoked and scaled. The word serverless isn't a great term for the technology, as the code is still run

on a server provided by Amazon, Microsoft, Google, or another provider. The difference is that the cloud provider handles the server for you, and they execute your code on the servers that they maintain and upgrade [39].

AWS provides several services in support of serverless. The primary service is Lambda, which was discussed in chapter 2. Lambda lets developers execute code without worrying about servers. Other services that are commonly used in serverless architectures are S3, DynamoDB, and API Gateway, all of which are utilized in this project. Some major companies that use AWS for serverless applications include Coca Cola, iRobot, Autodesk, and Square Enix [41].

3.3.2 Project Workflow

Before discussing the overall architecture of the project, it is important to understand the overall workflow from both the user perspective, as well as the admin perspective. The end user is going to interact with a web or mobile application, which in this project is coded using the React framework, but it could be any web or mobile application. It is important to remember that the real product here is the APIs. The front end is mostly for demonstration of the API's functionality. The front-end application communicates with the API by making a request to a specific endpoint and using a specific method, providing an API key in the header of the request, and consuming the response. The application can make GET requests to the services endpoint to see which third party face processing APIs are used in the project. The application can make POST and GET requests to the gladiator endpoint to submit images and obtain the results. Behind the scenes, the POST to the gladiator endpoint is storing the submitted image and

sending a notification via SNS to a Lambda for each third-party API. That Lambda then waits for the response from the third party and saves it to the database. The user then has to poll the gladiator endpoint using a GET request to get the accumulated results. The last endpoint that the application will hit is the bench endpoint, which will present aggregated results for each service from datasets that were run on the gladiator endpoint by the admin.

To look at the API from another perspective, let's consider a researcher who wants to use the API to compare which provider would give the best results for their dataset. They have no interest in using the front end. Their workflow would entail the following steps:

1) Upload images to S3 and make them public
2) Write a script to send the URLs of those images in the request body as a POST request to the gladiator endpoint
3) The gladiator POST request will return a link to /gladiator/{resultId} which the script should then poll until all services return a complete or failed status.
4) The user should then accept the GET request a write it to a file for further processing.

The architecture for the workflow of the API suite is shown in Figure 23.

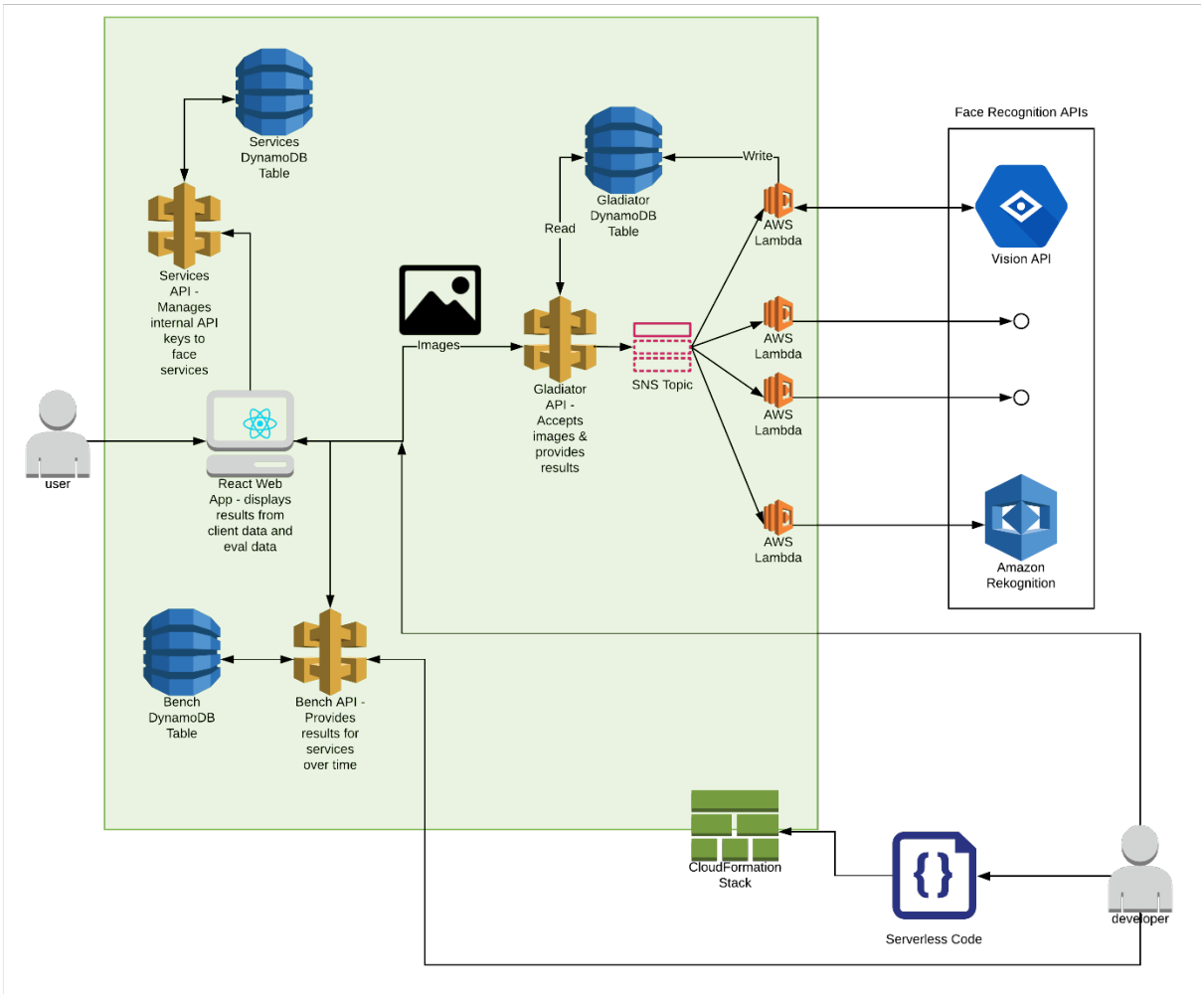


Figure 23 - Overall AWS architecture, discussed in 3.3.2

3.3.3 Serverless Framework Revisited

As mentioned earlier, the Serverless Framework creates the AWS resources necessary for a serverless application. In the serverless definition file, you specify the different Lambda functions that you want created, where the handler code is, and which events the function should respond to. The framework will create Lambdas for each definition and configure API gateway to point to each corresponding Lambda for each endpoint and method. This is exactly what is done for each user facing endpoint supplied

by MetaFace. The Lambdas that handle sending requests and accepting responses from the third-party face processing APIs are triggered from an SNS notification, which the gladiator POST endpoint sends information to. An excerpt of this configuration is shown in Figure 24.

```
rekognition:
  handler: src/function/serviceLambdas/rekognition.handler
  timeout: 30
  package:
    include:
      - src/function/serviceLambdas/rekognition.js
  events:
    - sns: ${self:custom.snsTopicName}

azure:
  handler: src/function/serviceLambdas/azure.handler
  timeout: 30
  package:
    include:
      - src/function/serviceLambdas/azure.js
  events:
    - sns: ${self:custom.snsTopicName}

watson:
  handler: src/function/serviceLambdas/watson.handler
  timeout: 30
  package:
    include:
      - src/function/serviceLambdas/watson.js
      - node_modules/watson-developer-cloud/**
  events:
    - sns: ${self:custom.snsTopicName}
```

Figure 24 - Serverless definition of Lambda functions that interact with third party APIs

The Serverless framework also allows developers to create other resources using CloudFormation syntax. In this project, DynamoDB NoSQL tables and the SNS topics are created using this functionality. Everything about the DynamoDB table that can be

configured in AWS can be configured through serverless. An example of this is shown in Figure 25.

```
resources:
  Resources:
    gladiatorTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${self:custom.gladiatorTableName}
        AttributeDefinitions:
          - AttributeName: resultId
            AttributeType: S
          - AttributeName: serviceId
            AttributeType: S
        KeySchema:
          - AttributeName: resultId
            KeyType: HASH
          - AttributeName: serviceId
            KeyType: RANGE
        ProvisionedThroughput:
          ReadCapacityUnits: 3
          WriteCapacityUnits: 3
```

Figure 25 - An excerpt from the serverless definition file that creates DynamoDB tables

Other features that the Serverless Framework offers include the ability to generate API keys and designate usage plans for each one, which allow you to limit the total requests per month and number of requests per second. You can also define the IAM policies that allow resources to request and send information to other resources. Lastly, a serverless plugin, named serverless-domain-manager, allows you to create and link a domain name to an API using Route 53 [40]. In this project, the API is linked to dev.gladiatorapi.com.

3.4 Database Design

NoSQL databases are a common choice for many serverless applications. They store data as key-value pairs, which is exactly how information is input and output from APIs. This especially makes sense considering the database needs to store varying amounts of information for each third-party API response. NoSQL databases are also cheaper than SQL databases, because the underlying engine is much lighter and, thus, they require less computational overhead [43].

Choosing tables for this project was not very difficult once the APIs were designed. Each endpoint gets their own NoSQL table. So, there is a bench, gladiator, and services table. Each table stores only information pertaining to its corresponding endpoint. Figure 26 shows the screenshot of the DynamoDB console, which lists each table and the unique keys that make up each record.

	Name	Status	Partition key	Sort key
○	bench-table-dev	Active	benchId (String)	-
○	gladiator-table-dev-a	Active	resultId (String)	serviceId (String)
○	services-table-dev	Active	serviceId (String)	-

Figure 26 - DynamoDB console of MetaFace tables

A partition key is like a primary key in SQL databases. It should be unique, unless a sort key is provided. A sort key allows duplicate values of a partition key, but only if they have different sort keys. It's DynamoDB's way of creating a composite primary key. As you can see in Figure 27, this methodology is used in the gladiator table. This makes retrieving each aggregated result much more efficient.

resultId	serviceId	imageUri	results	timest
195035bf-aedf-4947-a7b2-0b01dca477dc	1	https://s3.amazonaws.com/gladiator-images/HRT2...	{ "FaceDetails" : { "L" : [{ "M" : { "AgeRange" : { "M" : { "High" : { "N" : "43...	2019-0
195035bf-aedf-4947-a7b2-0b01dca477dc	2	https://s3.amazonaws.com/gladiator-images/HRT2...	[{ "M" : { "faceAttributes" : { "M" : { "accessories" : { "L" : [ ] }, "age" : { "N...	2019-0
195035bf-aedf-4947-a7b2-0b01dca477dc	3	https://s3.amazonaws.com/gladiator-images/HRT2...	{ "images" : { "L" : [{ "M" : { "faces" : { "L" : [{ "M" : { "age" : { "M" : { "ma...	2019-0
195035bf-aedf-4947-a7b2-0b01dca477dc	4	https://s3.amazonaws.com/gladiator-images/HRT2...	{ "image" : { "M" : { "height" : { "N" : "720" }, "orientation" : { "N" : "1" }, "w...	2019-0
195035bf-aedf-4947-a7b2-0b01dca477dc	5	https://s3.amazonaws.com/gladiator-images/HRT2...	{ "images" : { "L" : [{ "M" : { "faces" : { "L" : [{ "M" : { "attributes" : { "M" : ...	2019-0
195035bf-aedf-4947-a7b2-0b01dca477dc	6	https://s3.amazonaws.com/gladiator-images/HRT2...	{ "code" : { "N" : "400" }, "details" : { "M" : { "face" : { "M" : { "closedeyes" ...	2019-0
267d7e41-f0f9-4731-92c1-8a9c136d8340	1	https://s3.amazonaws.com/gladiator-images/HRT2...	{ "FaceDetails" : { "L" : [{ "M" : { "AgeRange" : { "M" : { "High" : { "N" : "15...	2019-0

Figure 27 - Screenshot of the Gladiator table console

Whenever the POST call is made to the gladiator endpoint. The gladiator Lambda function sends a notification to six Lambdas, one for each third-party API. These Lambdas send the request body corresponding to that third-party service, await the response, and write it to DynamoDB. The results from each of the six services need to be linked somehow. This is where the resultId, or partition key, comes into play. That resultId is the ID that is used in the GET request to Gladiator in order to obtain the collected results. Because that ID is also the partition key in the table, the lookup is optimally fast. The sort key tells which service the results came from. There is also a field for the image URL, the results returned from the service, and a timestamp to tell when the results were saved.

serviceId	endpointUri	lastUpdated	serviceName	key	appId
1	https://rekognition.us-east-1.amazonaws.com	2019-01-15T19:47:51.090Z	Rekognition		
2	https://eastus.api.cognitive.microsoft.com/face/v1.0/detect	2019-01-15T19:47:51.090Z	Azure	8609955...	
3	https://gateway.watsonplatform.net/visual-recognition/api	2019-01-15T19:52:33.710Z	IBM Watson	E72QPX...	
4	https://prod.sighthoundapi.com/v1/detections?type=face&fa...	2019-01-21T01:30:08.823Z	Sighthound	T27VtJ5b...	
5	https://api.kairos.com/detect	2019-02-11T13:50:28.160Z	Kairos	f8481e4c...	c7d...
6	https://chronos.lapetussolutions.com/v2/o/i3s/a/metaface/us...	2019-02-18T16:01:23.766Z	Lapetus	fo2q2Qa...	

Figure 28 - Screenshot of the Services table endpoint

Figure 28 is a screenshot of the services table. There are only 6 entries in the table at the time of writing, and they contain all of the information necessary to make requests to each API. Because of the NoSQL design, the extra field that the Kairos service requires, appId, was added with ease. You can also see that AWS Rekognition has no key because the API is accessed with inter-resource communication through IAM policies.

3.5 API Requests and Response Demonstration

In this section, examples of requests and responses are demonstrated (Figures 29-31). All of the requests are made using Postman, which is a popular tool for testing APIs. Postman offers a lot of features for building and testing APIs; however, it is only used for sending manual requests to APIs in this context [42]. The responses from these endpoints reflect the structure that was defined in the API documentation and return the information that is stored in the database.

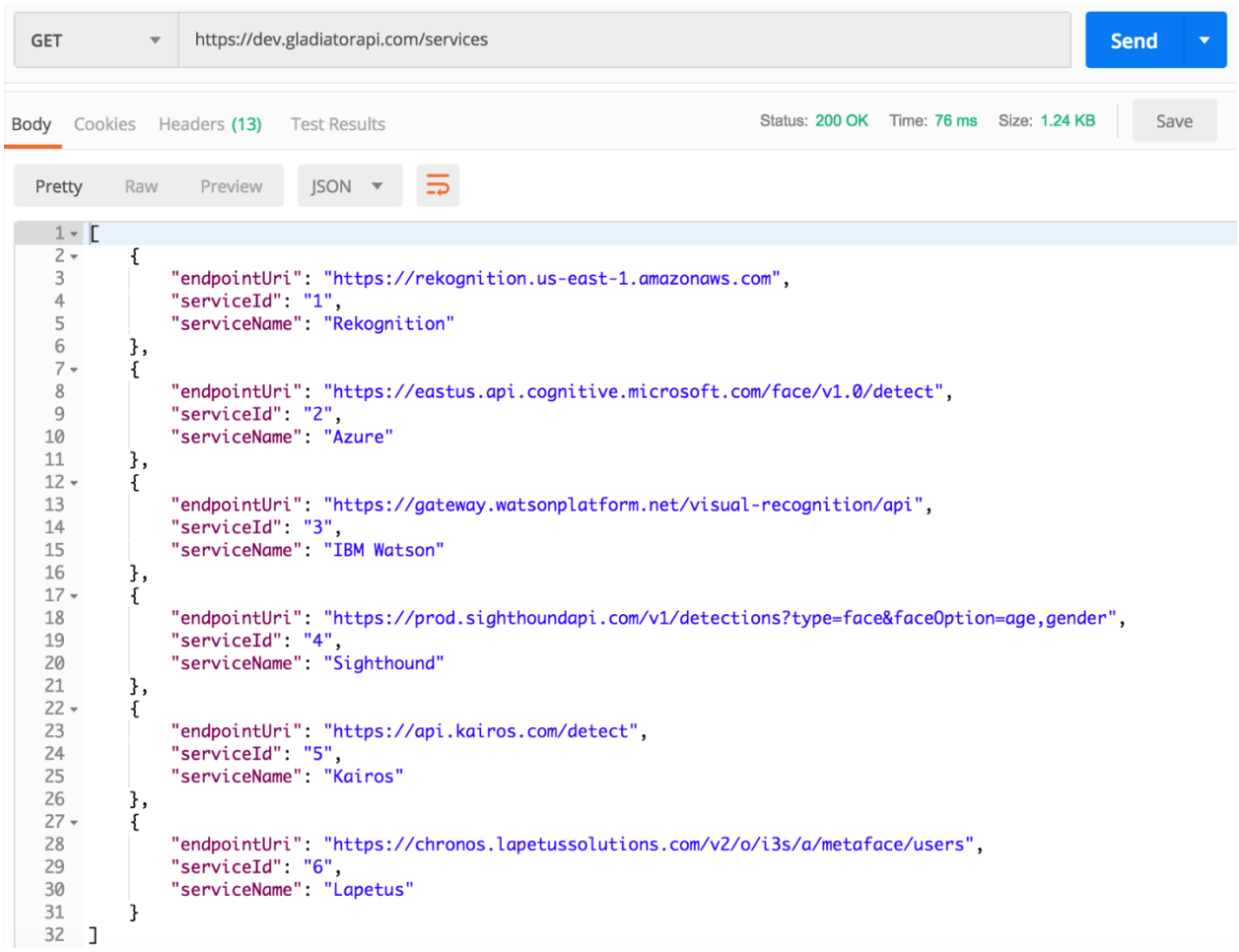


Figure 29 - A sample response body from the GET services endpoint

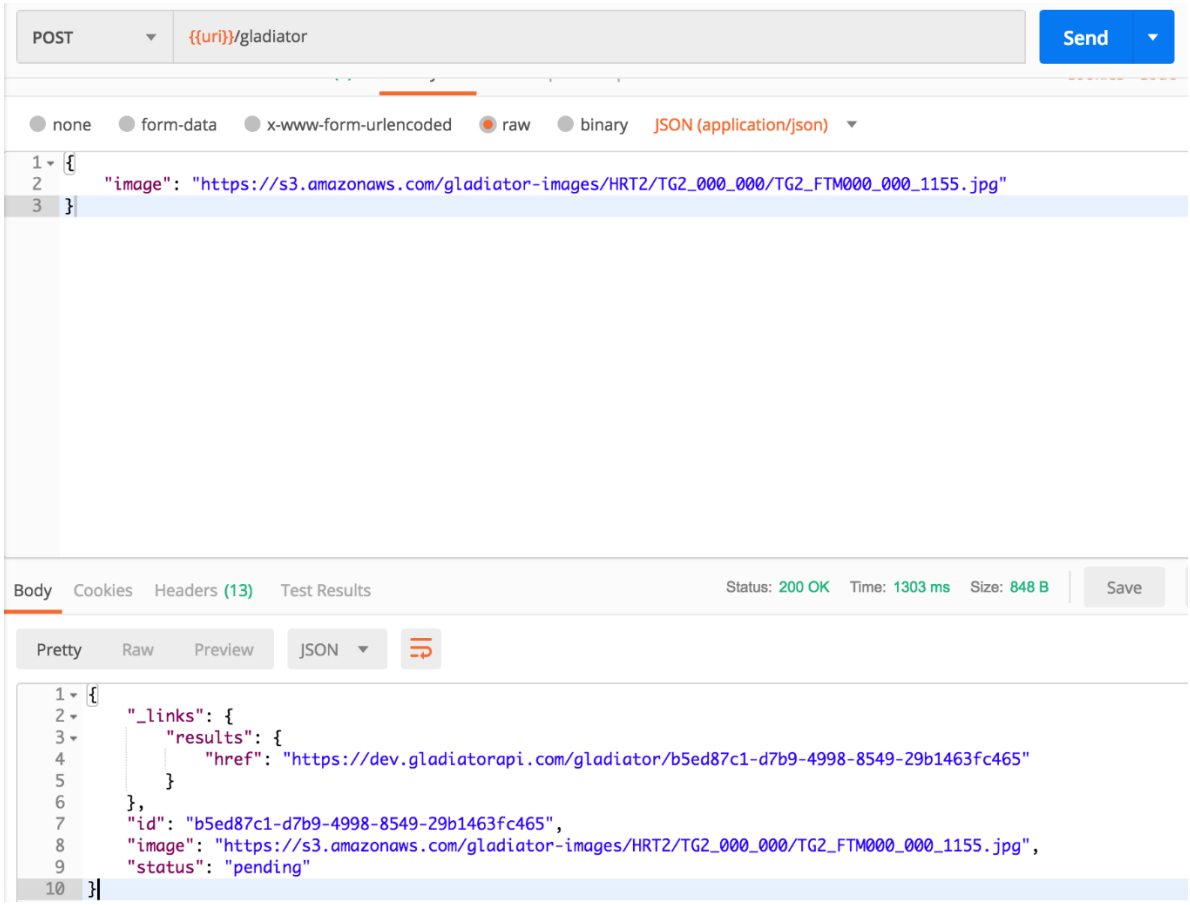


Figure 30 - A sample request and response body from the POST gladiator endpoint

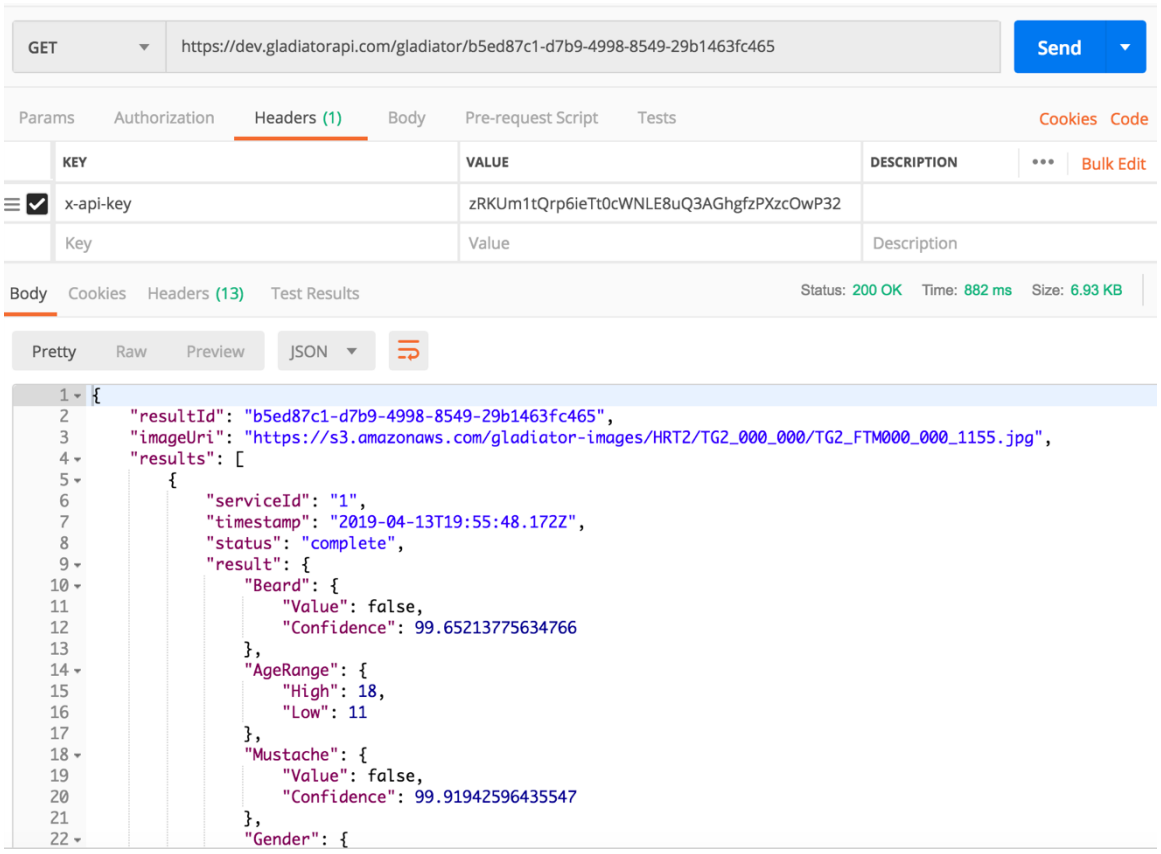


Figure 31 - A sample request header and response body from the Gladiator GET endpoint. The response body is about 600 lines long, and it contains results from 6 services.

3.6 Front-End Design and Demonstration

To demonstrate this project's API in a user-friendly way, a front-end web application was created. The goal of this application was to allow developers to upload an image and view the results from the gladiator endpoint in a visual way. Another goal of the application was to graphically view how each third-party face processing API performs for each age and gender.

3.6.1 React

With all of the major frameworks that have been demonstrated in this project, it is no surprise to learn that many front-end frameworks exist for Node environments. The most common current design pattern for web applications is called single-page. When navigating through most modern websites, you will notice that web pages no longer perform full refreshes when going from one URL to another. The concept of single-page applications says that only part of the web page will be updated for each browser event. For instance, a navigation menu bar should not need to be re-rendered for each tab that is clicked [55].

React, a front-end framework that was released and is still maintained by Facebook, was created to allow developers to design and implement single page apps. React is used for both Facebook and Instagram, which both have web applications as well as mobile applications, so you know it can support applications with millions of users. React organizes of the different function points of a web application into components. A component can be as granular as a text box, or as complex as an entire form entry. Generally, a component should be small enough such that it can be rendered independently from other pieces of a website. When using react properly, every web page is dynamically rendered such that each component is rendered as they're loaded [56].

3.6.2 Components

Because React is so popular, there are many libraries that provide pre-designed components for React. These component libraries provide UI elements for navigation

bars, buttons, images, cards, tables, and more. Two major component libraries that are utilized in the MetaFace front end are material-ui and react-bootstrap.

Material-ui provides React components that implement Google's material design. These are the buttons, forms, and other components that Google created for all of their web pages. They are very responsive and include some eye-catching animations even for simple components like buttons [57].

Bootstrap has been responsible for many progressive web application designs since 2011. Bootstrap was used to solve the problem that many websites had with dynamically rendering their web pages for phones and browsers viewed on larger computer screens. Bootstrap is best suited for mobile-first development, where web pages are designed for mobile screens first [58]. React-bootstrap provides classes, including React components, for implementing bootstrap's popular grid layout and UI design [59]. In this project, material-ui was primarily used for buttons, forms, displaying images while bootstrap was used for the navigation bar and the layout functionality. With the combination of these two tools, the application is well organized and can be viewed from browsers and phones.

3.6.3 Amplify

Amplify was introduced in Chapter 2. Amplify is an AWS-provided SDK for performing common AWS operations from the front end [40]. In this project, the React application makes API calls using Amplify. These are organized into reusable classes for each API endpoint that wrap Amplify's API package. Amplify is also used to interact

with AWS Cognito for user authentication and registration. Amplify controls all of the interaction with Cognito to store usernames, passwords, and the user's API key!

3.6.4 Application Features

Using the MetaFace web application, users can register for an account by providing an email, password, and API key for Gladiator. Currently, this API key must be supplied by an admin, as API key registration will be implemented whenever a billing workflow is setup. Once the user completes the sign-up form, an automated email is sent to them with a confirmation code. This confirmation code is then entered on the application, which grants the user access to their account. The application contains a tab for gladiator, bench, settings, and logout. The settings tab allows a user to change their email, password, or API key. These are all integrated with Cognito (through Amplify) such that the user information is updated on Cognito as well as the user's browser session cache. The gladiator tab allows an end user to upload an image and the results are displayed in a list format on the gladiator main page. The list displays the image name, when the image was submitted, and a thumbnail of the image. When the user clicks on a list, a page is loaded that displays the following:

- The image
- A table which displays the response from gladiator
- A button for the user to download the JSON representation of the results
- A drop-down button that displays the entire JSON result in a formatted HTML template which allows them to collapse and expand certain fields of the response

The Bench tab currently displays some graphs for the result sets that are returned from the Bench endpoint. These graphs are not pre-generated but rendered using data points as part of the application. An important feature of this application is that each user sees only the images and results for the images that they've submitted. So, you won't see results that another user has requested! To see screenshots from the web application, see Appendix C.

3.7 Unit Testing Results

There are 42-unit tests written for this project. Each of the endpoint's express modules were tested and checked for code coverage. The tests cover all of the error response branches and the successful responses, as well as the different parsing scenarios for each third-party service response. Figures 32 and 33 show the overall coverage for each module/endpoint. For an example of how code coverage can be used to generate heat maps that show which code is not covered by tests, see Appendix D.

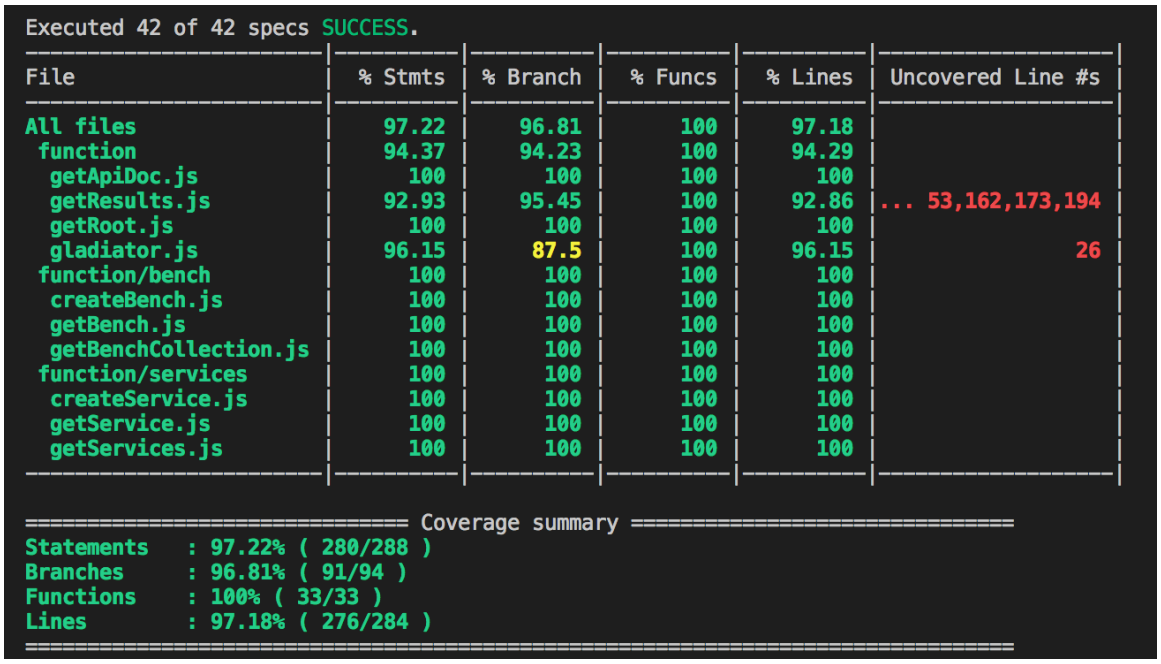


Figure 32 - Screenshot of code coverage console print out

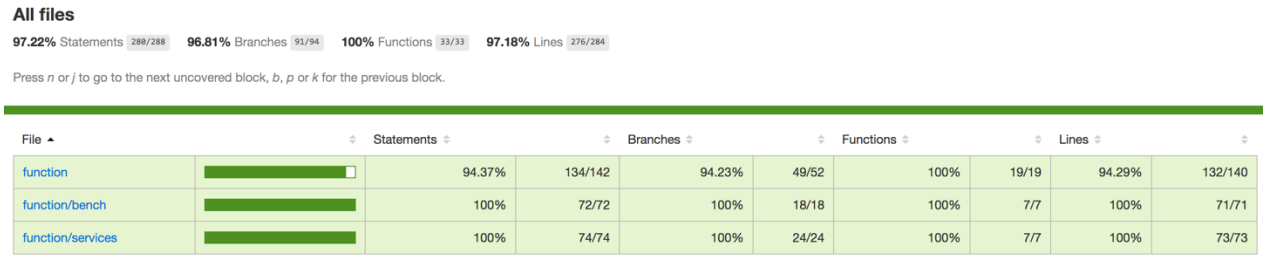


Figure 33 - Screenshot of overall coverage in nyc UI

CHAPTER 4: RESULTS OF COMPLETED PROJECT

4.1 Products of the Project

The previous chapters detail what different components of the API product were produced, and which tools, technologies, and methodologies were used to create them. In this section, all of the different components of the project will be summarized.

The API design was documented and published so that other developers can easily interact with the API and use them for comparing results from multiple face processing APIs. The documentation template can be generated and deployed within seconds, which makes the documentation robust for changing as the API implementation changes.

An API suite which contains the user-facing endpoints gladiator, bench, and services were created, implemented, and deployed onto AWS. The backend of the APIs is all handled using Lambda which implements the serverless architecture. This saves money and maintenance requirements because servers do not have to be provisioned, persisted, managed, and upgraded.

The Gladiator endpoint accepts a user image, and concurrently sends it to numerous third-party face processing APIs. The Gladiator endpoint can then be polled using GET requests until all of the third-party APIs return a response to Gladiator, which are then returned to the user with a "complete" status. Gladiator is the center piece of the entire product.

The Bench endpoint allows admins to create benchmarking information for a given dataset. Currently, this benchmarking information includes how each third-party

API performs for each age and gender. The graph from these results are presented and discussed in the next section.

Lastly, the services endpoint stores all of the information for each third-party API. This includes the URL, API key, and service name. The services endpoint can be accessed to see which serviceId maps to which service name. It is also used by the Gladiator endpoint to retrieve the API key for each service.

Another product of this project is the MetaFace web application. This product suite contains the Gladiator and Bench products and allows users to interact with them through a front-end application. This website allows users to register for and login to the application using AWS Cognito for authentication.

4.2 Benchmark Results

The primary purpose of this project is to provide a way for researchers and companies to evaluate each face processing API for their application use case. The best way to test the business case of Gladiator is to feed it a dataset that has known ages and genders and see if the responses from each service can be compared.

Figure 34 displays that the only overlapping fields that are returned from all services are age and gender. For this reason, only age and gender were compared for this sample benchmark. That being said, Gladiator could be extended to compare more fields

from a subset of the services offered below. Gladiator also allows new services to be added within minutes, so their results could be compared as well.

Service	Response Includes
AWS Rekogntion	Age (range), gender, facial hair, glasses, emotions, landmarks, eyes/mouth open, smile,
Microsoft Azure	Age, gender, bounding box, facial hair, hair color, emotion, makeup, landmarks
IBM Watson	Age (range), gender, bounding box
Slghthound	Age, gender, bounding box
Kairos	Age, gender, bounding box, glasses, mouth open, yaw, pitch, roll, ethnicity, some landmarks
Lapetus	Age, gender, BMI, smoker

Figure 34 - List of services and the fields they return

The dataset that was chosen for a sample benchmark is the UTKFace dataset [54]. This dataset contains thousands of images where the age and gender of the person in the image are known and provided. The images do not have a very high resolution so there were many images where every service could not find a face in the image. To evenly compare how each service performs on identifying a subject's age and gender, I removed any result set that did not contain results from each service. The point of this benchmark sample was not to see which service identified faces in an image the best, but to find which service was most accurate in predicting age and gender when a face was found. After removing incomplete result sets, the results from about 3800 images were left. The

graphs from the results are identified using the service ID. For reference, the mapping of service IDs to service names is shown in Figure 35.


```
[
    {
        "endpointUri": "https://rekognition.us-east-1.amazonaws.com",
        "serviceId": "1",
        "serviceName": "Rekognition"
    },
    {
        "endpointUri": "https://eastus.api.cognitive.microsoft.com/face/v1.0/detect",
        "serviceId": "2",
        "serviceName": "Azure"
    },
    {
        "endpointUri": "https://gateway.watsonplatform.net/visual-recognition/api",
        "serviceId": "3",
        "serviceName": "IBM Watson"
    },
    {
        "endpointUri": "https://prod.sighthoundapi.com/v1/detections?type=face&faceOption=age,gender",
        "serviceId": "4",
        "serviceName": "Sighthound"
    },
    {
        "endpointUri": "https://api.kairos.com/detect",
        "serviceId": "5",
        "serviceName": "Kairos"
    },
    {
        "endpointUri": "https://chronos.lapetussolutions.com/v2/o/i3s/a/metaface/users",
        "serviceId": "6",
        "serviceName": "Lapetus"
    }
]
```


Figure 35 - Mapping from service IDs to names

The first comparison that was made was identifying which service predicted gender most accurately (Figure 36). This comparison was quite easy, as each service only returns one value for gender and there are only two classes returned: male and female. Every service predicts gender at least 85% of the time, while Azure and Kairos have the best accuracy for the UTKFace dataset. It is important to note that the ranking would likely be different for a dataset which has higher resolution images or images with different lighting conditions.

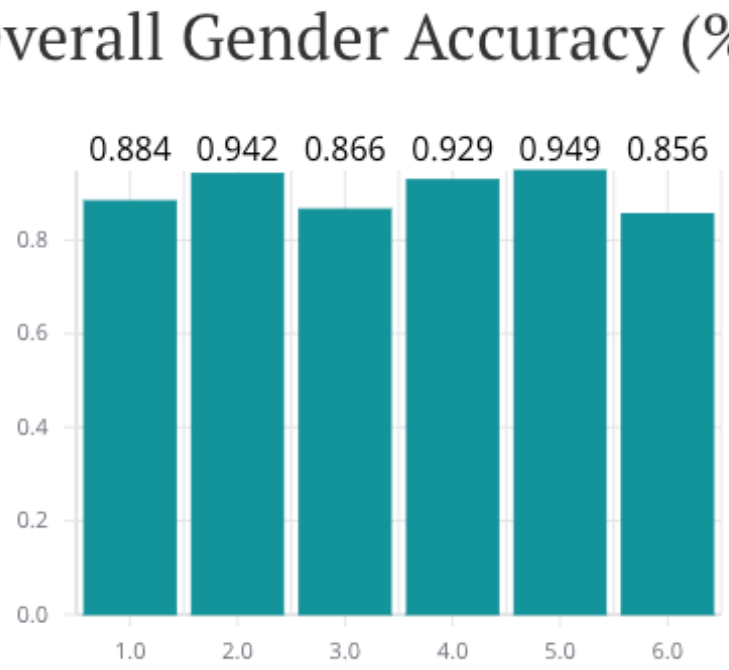


Figure 36 - Overall accuracy for gender

The other field that was compared between APIs is age. Age was much more complex to compare, because two third-party APIs return a range of numbers whereas four return a single value estimate. The means for comparing single value estimates from a range value is much different, so the comparison between these two different response types were split up. The two graphs below (Figures 37 and 38) show the overall results for each response type. The services that return a range of values for age are scored based on whether the subject's age fell between the predicted range or not. The services that return a single value estimate are evaluated based on the average error between the predicted and actual value. There are 7 classes for age because Lapetus returns two different predictions for age, based on different algorithms. These two algorithms are represented in class 6 and 7.

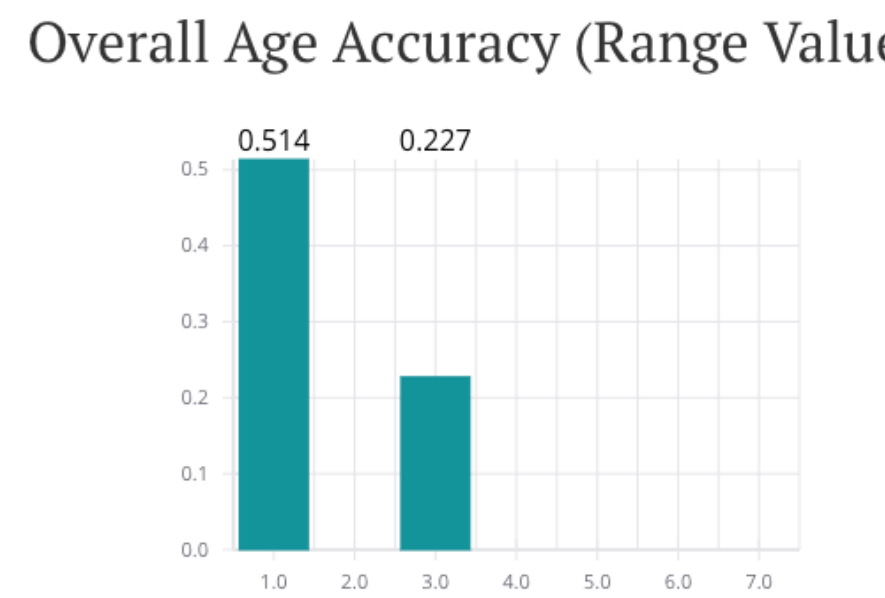


Figure 37 - Overall accuracy for age (range values)

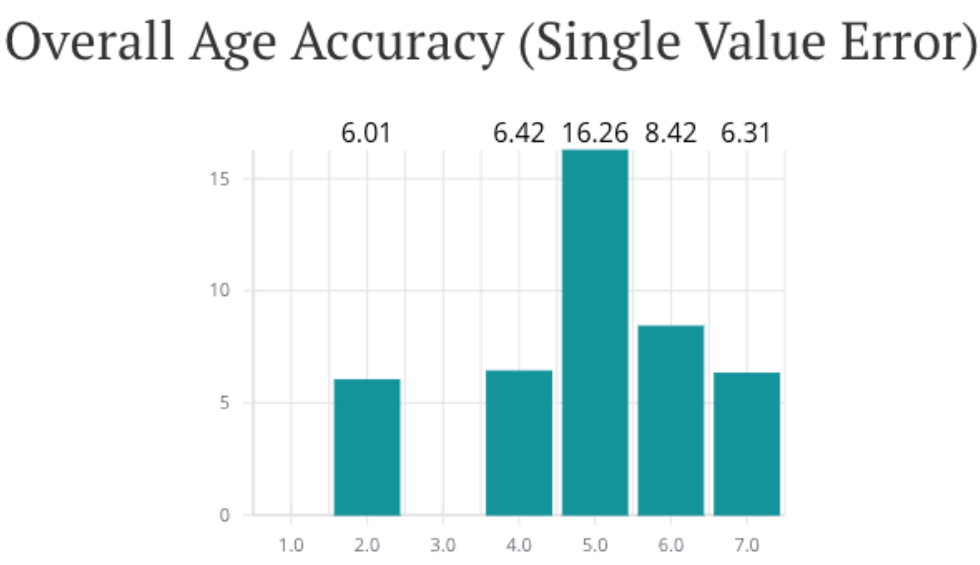


Figure 38 - Overall accuracy for age (single value estimates)

The next graphs that are presented show each service's accuracy for each age value. The first two graphs show the accuracy for the services that return age as a range, Rekognition and Watson (Figures 39 and 40). Two important discoveries can be made from these graphs. You will notice that Rekognition does not predict high values over 90 for age and Watson does not return values over 75 as the high end of the range. With discoveries like these, companies could decide not to use Watson because they need to predict ages over 76.

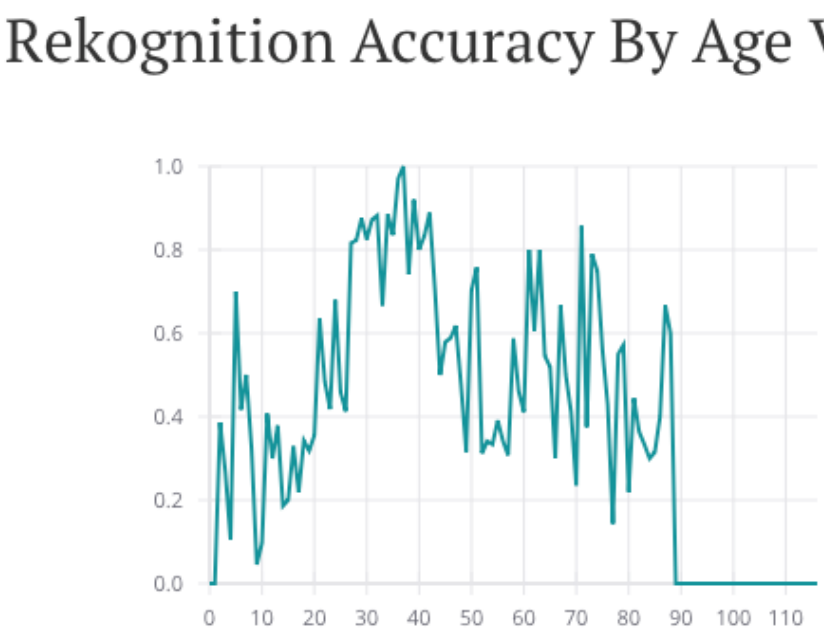


Figure 39 - AWS Rekognition accuracy for each age value

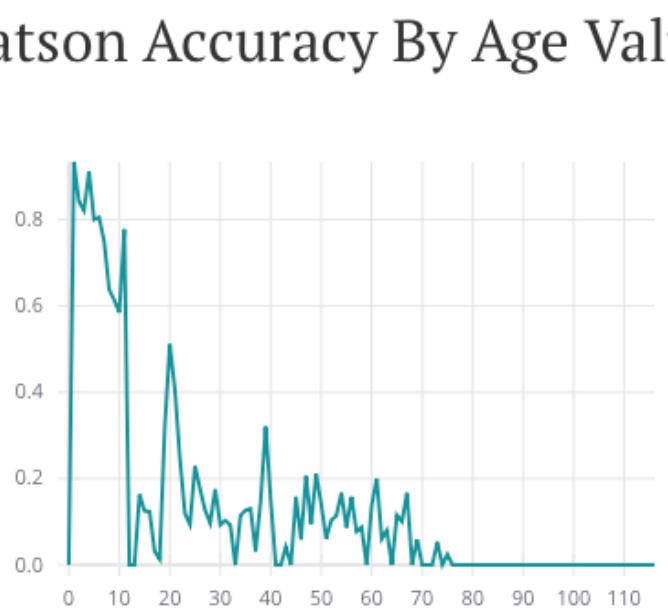


Figure 40 - IBM Watson accuracy for each age value

The graphs below (Figures 41 – 45) show the accuracy ratings for services that return age as a single value. The comparisons were made using the same method as for the overall results. So, this means each data point shows the average error for every image of each age. For the graphs below, it is important to note that an accuracy value of 0 does not mean that the average error is 0. It means that there is not an image in the dataset that has a subject with that age. You would not expect algorithms to perform as well on higher ages because there are less subjects to train algorithms on (i.e. there are very few images of people over 110). The graphs backup that hypothesis because services rarely predict ages over 90.

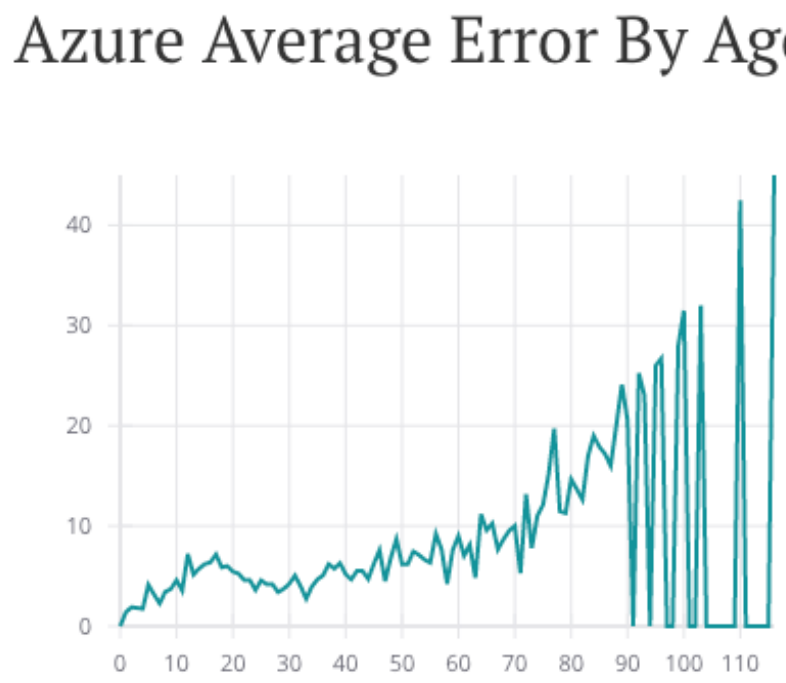


Figure 41 - Microsoft Azure accuracy for each age value

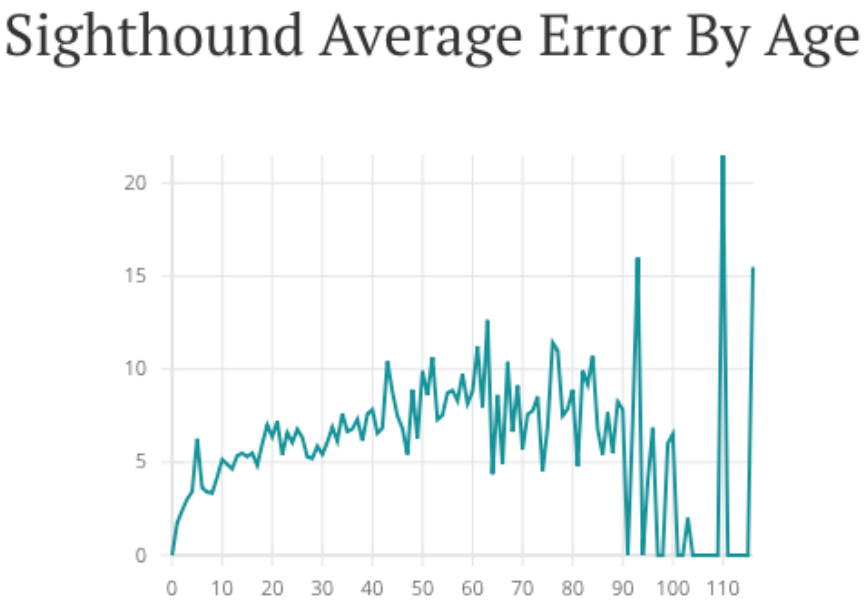


Figure 42 - Sighthound accuracy for each age value

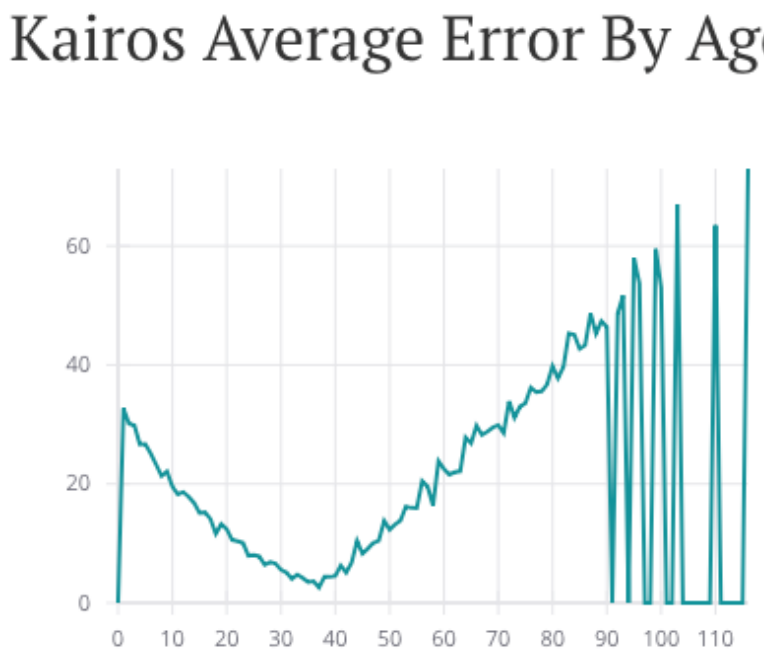


Figure 43 - Kairos accuracy for each age value

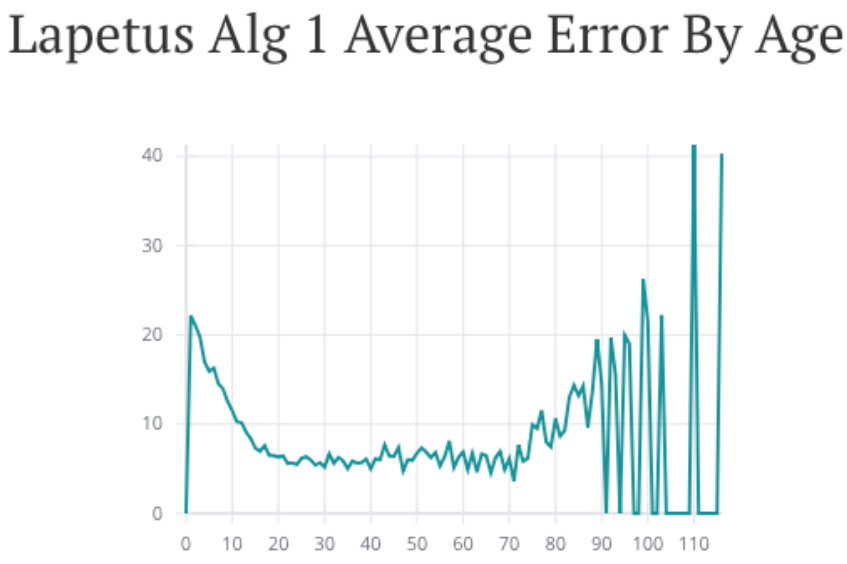


Figure 44 - Lapetus algorithm 1 accuracy for each age value

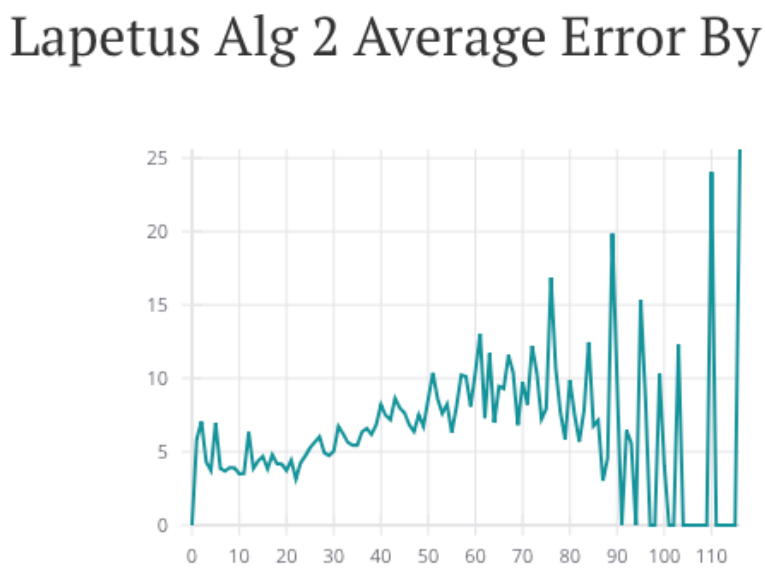


Figure 45 - Lapetus algorithm 2 accuracy for each age value

4.3 Sample Results

In this section, some sample responses from the Gladiator endpoint are shown in Figures 46 – 49. They are displayed from the MetaFace application individual result page. All of the images shown in the sample responses are from the public dataset, UTKFace. The subject's age and gender can be found in the image name of the result set. The naming format for UTKFace in MetaFace is timestamp-age_gender_ethnicity_dateOfAcquisition.jpg.chip.jpg, where gender maps to 1 for a female and 0 for a male. For more example results, see Appendix E.

Gladiator / 1556163082649-10_0_0_20170103200522151.jpg.chip.jpg

Service ID	Status	Age	Gender
1	complete	{"High":18,"Low":11}	Female
2	complete	6	male
3	complete	{"max":12,"min":0}	female
4	complete	5	female
5	complete	34	M
6	complete	34.4	male

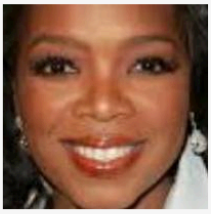
Figure 46 - Example Gladiator response for a 10 year old male

Gladiator / 1556426207763-30_1_0_20170103182425185.jpg.chip.jpg

Service ID	Status	Age	Gender
1	complete	{"High":43,"Low":26}	Female
2	complete	31	female
3	complete	{"max":32,"min":28}	female
4	complete	29	female
5	complete	29	F
6	complete	32.6	female

Figure 47 - Example Gladiator response for a 30 year old female

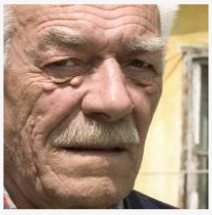
Gladiator / 1556425874487-52_1_1_20170113011248822.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":43,"Low":26}	Female
2	complete	36	female
3	complete	{"max":41,"min":38}	female
4	complete	33	female
5	complete	39	F
6	failed	-	-

Figure 48 - Example Gladiator response for a 52 year old female

Gladiator / 1556160354603-79_0_0_20170111210813495.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":80,"Low":60}	Male
2	complete	71	male
3	complete	{"max":12,"min":0}	female
4	complete	88	male
5	complete	43	M
6	complete	70.3	male

Figure 49 - Example Gladiator response for a 79 year old male

4.4 Discoveries

A few things were discovered about how each API differs, including how they accept requests and return responses. All of the APIs have a capacity for image size, but they all differ. Rather than set a maximum image size capacity on the Gladiator API, I allowed all images and present the error from any service that returns an error. Some APIs also have format restrictions, where they may not accept GIF formatted images. All of the services in this project allow jpg, jpeg, and png, which are the most common image formats. Another discovery is that half of the APIs will not identify faces in images that are rotated 90, 180, or 270 degrees.

A big challenge for implementing Gladiator was passing images in different formats for different services. Lapetus, for instance, takes a binary representation of an image as input, whereas the other services accept the image as a public URL. S3 served this use case well because images can be uploaded with a public URL, and the binary representation can be retrieved from that URL.

The APIs all return responses in different formats. The JSON structure of each response if different, which is to be expected, but even the genders are returned different strings. As seen in the example responses above, some return ‘M’ while others return ‘male’ or ‘Male’. These could easily be parsed at the API or web application level, but I wanted to leave them as is to display how different all of these APIs really are. A more complex response to handle was the difference between age ranges and single value estimates, which was discussed in an earlier section.

4.5 Problems

Many problems were encountered and overcome throughout this project, but I wanted to share some of the major ones. One such major problem were the default limits that are set by AWS. When I created the API keys, they are assigned to a default usage plan that restricts requests to 10,000 calls per month. I hit that restriction while running my script to evaluate the UTKFace dataset, and suddenly started getting 403 status codes for unauthorized requests.

Another major problem came with S3 pagination. S3 automatically paginates objects by the 1000, which also messed up my evaluation script when I was retrieving the 3800 images from UTKFace in S3. Another issue that came about with S3 is that images must be uploaded with a strict object policy for them to be accessed by the other services that accept images as URLs. The last major problem I encountered was hitting the open file limit on the Mac OS when deploying the entire stack to AWS. I had to alter some system configurations just to deploy the entire stack.

CHAPTER 5: CONCLUSIONS AND FUTURE WORK

5.1 Future Work

This project could be productized in its current form, but several improvements would need to be made to allow growth. The product would need an entire development team, so a develops pipeline would need to be created for concurrent development between teams. This pipeline would include multiple environments, i.e. sandbox, integration, production, and integration/system tests on pull requests. Other required features include:

- The processing of payments and storing of payment information
- A design for the web application with full user registration an API key retrieval
- The ability to upload and process entire datasets at a time within the application
- An API versioning strategy for future improvements and remediation
- The ability to keep user uploaded images private and give pre-signed temporary URLs to the third-party services

Gladiator could be created and marketed in a few ways. First off, the project could easily be configured to track the results of each service over time, which would provide insight on how often each company alters or improves their algorithm. The web application demo could be extended in two directions. The first could be a product that focuses on delivering the results from each service to end users. The second is less straightforward. An algorithm could be placed in the API that returns the best results for each service. For example, it returns gender from Kairos and age from Azure since they were the least erroneous for the UTKFace dataset. This logic

would obviously have to be more complex, but it is certainly possible to implement this solution.

5.2 Conclusion

The goal of this project was to deliver an API that provides the means to benchmark other multiple third-party face processing APIs. The Gladiator endpoint does just that. It aggregates the responses from infinitely many third-party APIs and presents them to a user in one place. They only have to manage a single API key and pay a single bill, and they get the power of several APIs. The Gladiator endpoint could link up to many more face processing APIs, but it could also utilize APIs of other types. Gladiator could one day be used to compare text recognition or license plate detection algorithms.

Other takeaways from this project include the in-depth API documentation that was written and published. This documentation enables other developers who may want to consume the Gladiator API for their applications, like the MetaFace application.

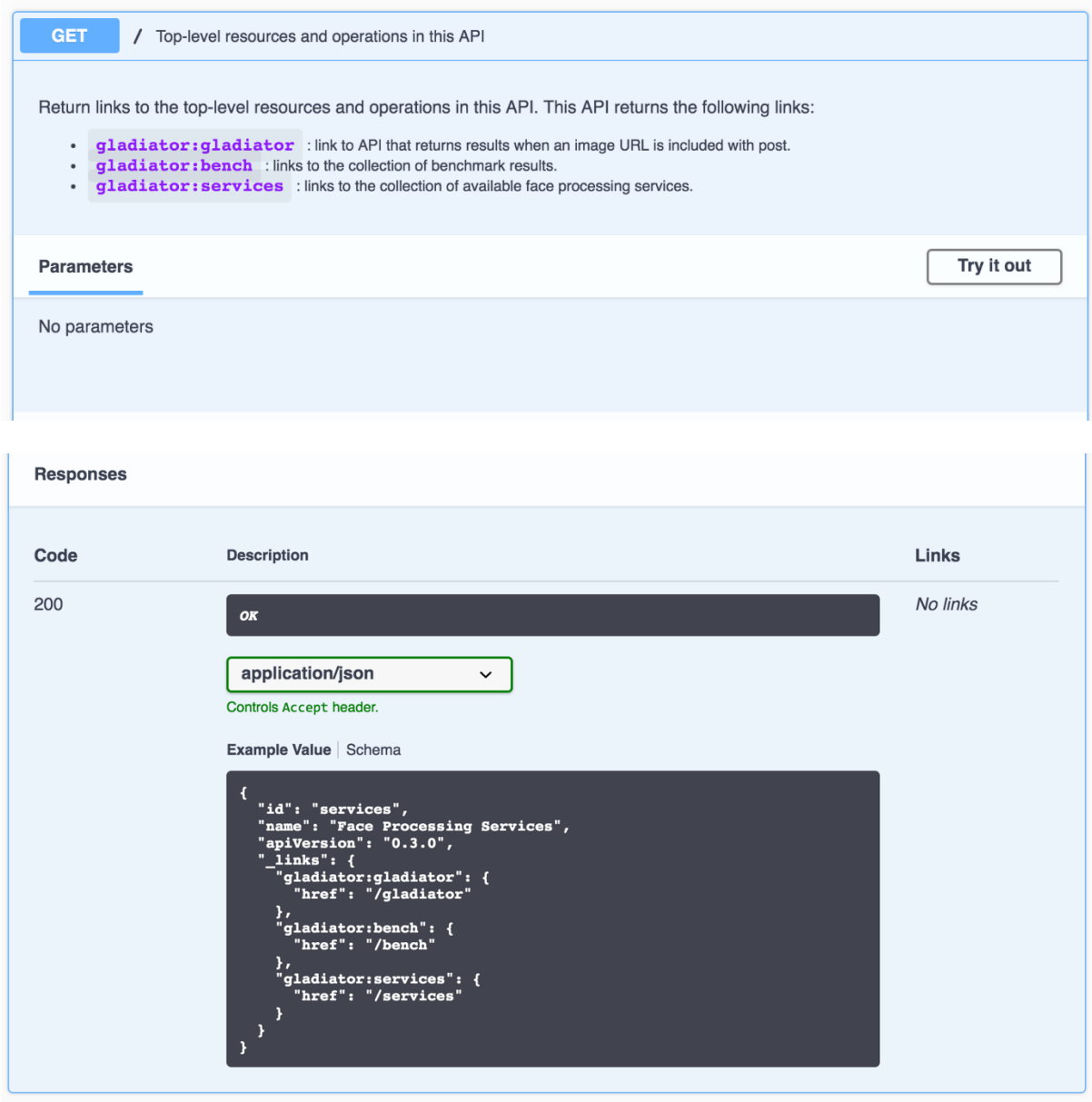
The MetaFace web application lets users register, login, and interact with the Gladiator API in a user-friendly way. With the application, non-API developers can use Gladiator for research or evaluation. The MetaFace also allows for user management and viewing results from the Bench API.

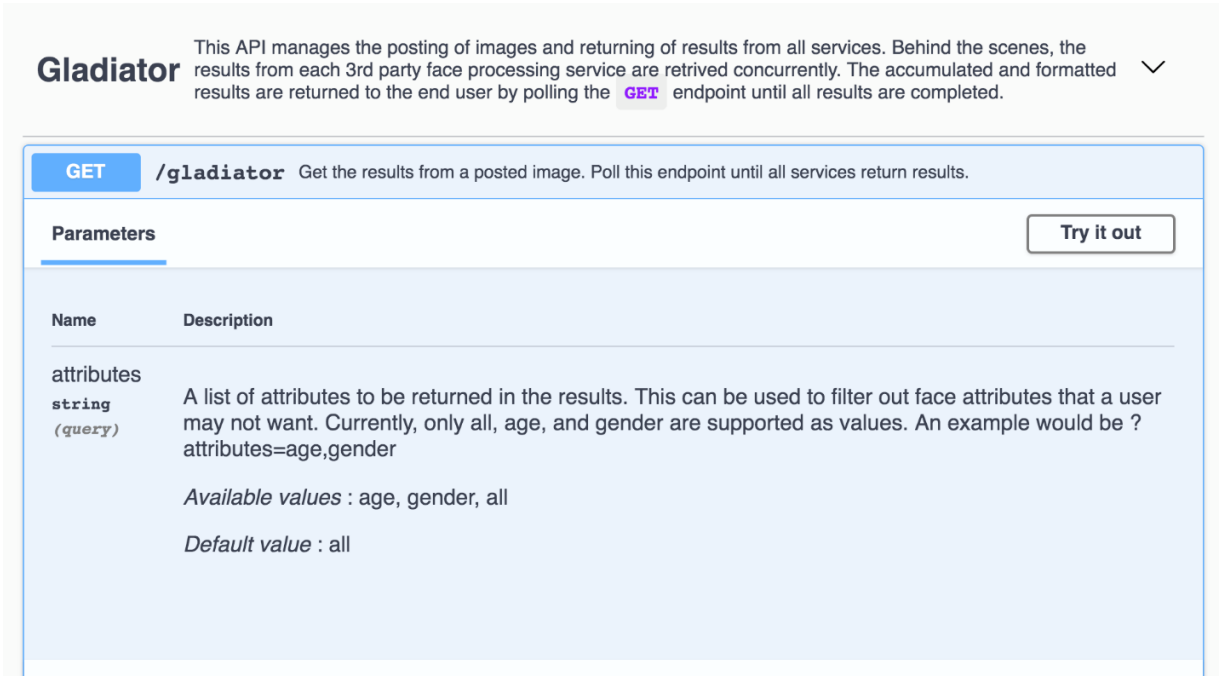
The last thing I want to mention is the reliability, security, scalability, and cost optimization that this project was built around. The entire project could be redeployed in 10-20 minutes in case of a disaster, and the database can be configured with automatic backups at any time interval. The API is secure through the use of API keys, and the web application is secure through the use of Cognito user and identity pools. Because the

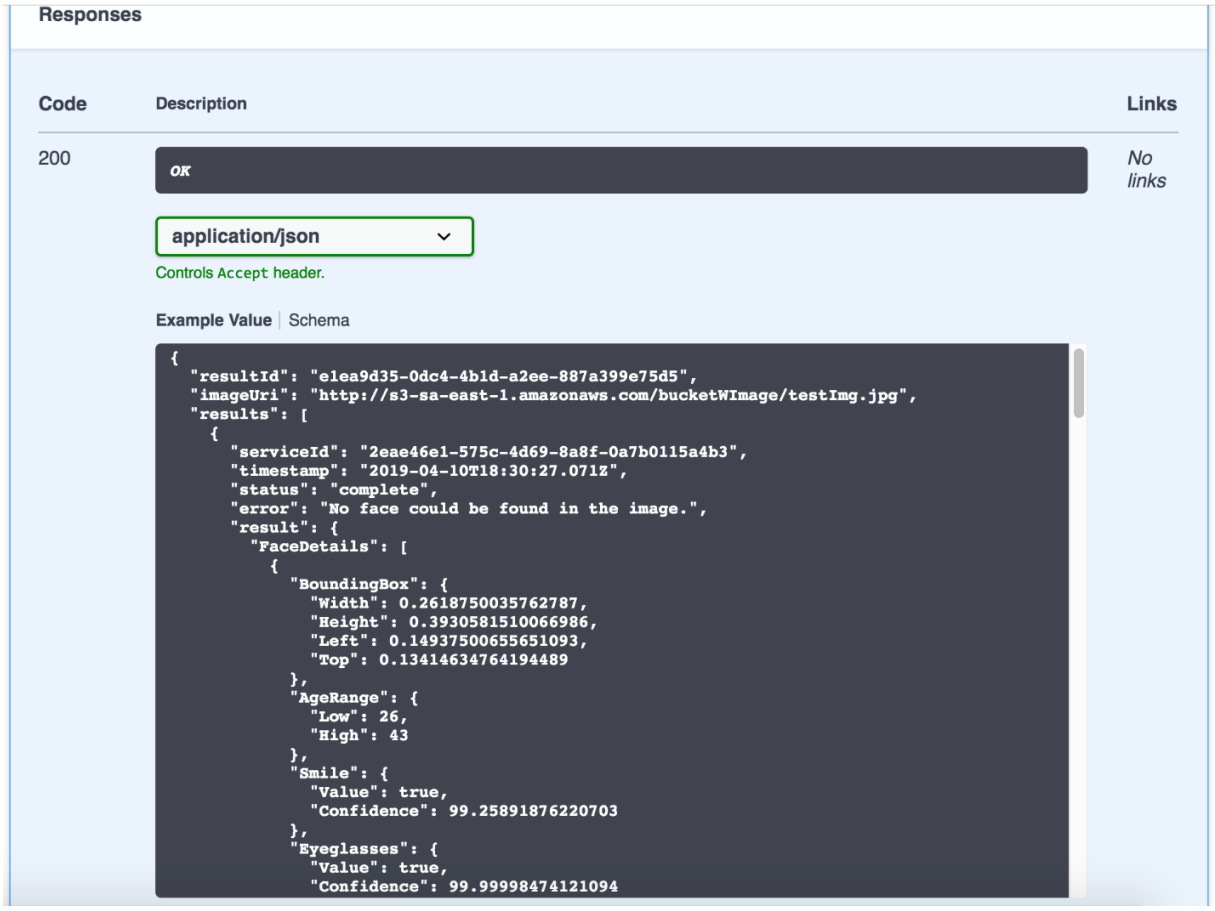
serverless architecture and AWS lambda were used, the API can scale to handle traffic from thousands of users concurrently. The design patterns used between the Gladiator and Services endpoint allows services to be added on to the project within minutes, so the API could grow to include hundreds of third-party APIs in many different problem areas. Gladiator gives researchers and companies, both API providers and consumers, the ability to discover how different APIs perform in different problem domains.

Development of this project and product will continue into the future. There is a business case for several components in this project, and it has already uncovered several fascinating discoveries in what kind of data companies use to train their face processing algorithms. This was a great start, but there are still many more discoveries to make.

Appendix A (Gladiator API Documentation Using Swagger)




Gladiator
This API manages the posting of images and returning of results from all services. Behind the scenes, the results from each 3rd party face processing service are retrived concurrently. The accumulated and formatted results are returned to the end user by polling the GET endpoint until all results are completed.
GET
/gladiator
Get the results from a posted image. Poll this endpoint until all services return results.
Parameters
Try it out
Name
Description
attributes
string
(query)
A list of attributes to be returned in the results. This can be used to filter out face attributes that a user may not want. Currently, only all, age, and gender are supported as values. An example would be ? attributes=age,gender
Available values : age, gender, all
Default value : all


Responses
Code
Description
Links
200
OK
No links
application/json
Controls Accept header.
Example Value
Schema
{
"resultId": "e1ea9d35-0dc4-4b1d-a2ee-887a399e75d5",
"imageUri": "http://s3-sa-east-1.amazonaws.com/bucketWImage/testImg.jpg",
"results": [
{
"serviceId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
"timestamp": "2019-04-10T18:30:27.071Z",
"status": "complete",
"error": "No face could be found in the image.",
"result": {
"FaceDetails": [
{
"BoundingBox": {
"Width": 0.2618750035762787,
"Height": 0.3930581510066986,
"Left": 0.14937500655651093,
"Top": 0.13414634764194489
},
"AgeRange": {
"Low": 26,
"High": 43
},
"Smile": {
"Value": true,
"Confidence": 99.25891876220703
},
"Eyeglasses": {
"Value": true,
"Confidence": 99.99998474121094

404	*Not found. There is no such service resource at the specified {resultId} the _error field in the response will contain details about the request error.*
	
	application/json
	
	Example Value | Schema
	
	{
	"_error": {
	"_id": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
	"message": "A resource cannot be found with the given userId",
	"statusCode": 404,
	"attributes": {
	"userId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b4"
	},
	"remediation": "Resubmit the operation with an existing userId",
	"occurredAt": "2018-01-25T05:50:52.375Z"
	}
	}
		No links

500	*Database or server error. Please wait a minute and retry request.*	*No links*

POST `/gladiator` Post an image to be run through third party face processing APIs

Parameters Try it out

No parameters

Request body required application/json

The user and image information required to get results

Example Value | Schema

```
{
  "resultId": "e1ea9d35-0dc4-4b1d-a2ee-887a399e75d5",
  "imageUri": "http://s3-sa-east-1.amazonaws.com/bucketWImage/testImg.jpg",
  "apiKey": "abcdef12345"
}
```

Responses

Code	Description	Links
200	*OK*	*No links*

400

Bad Request. Either the given image URL is invalid or missing.

No links

application/json

Example Value | Schema


```
{
  "_error": {
    "_id": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
    "message": "A resource cannot be found with the given userId",
    "statusCode": 404,
    "attributes": {
      "userId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b4"
    },
    "remediation": "Resubmit the operation with an existing userId",
    "occurredAt": "2018-01-25T05:50:52.375Z"
  }
}
```

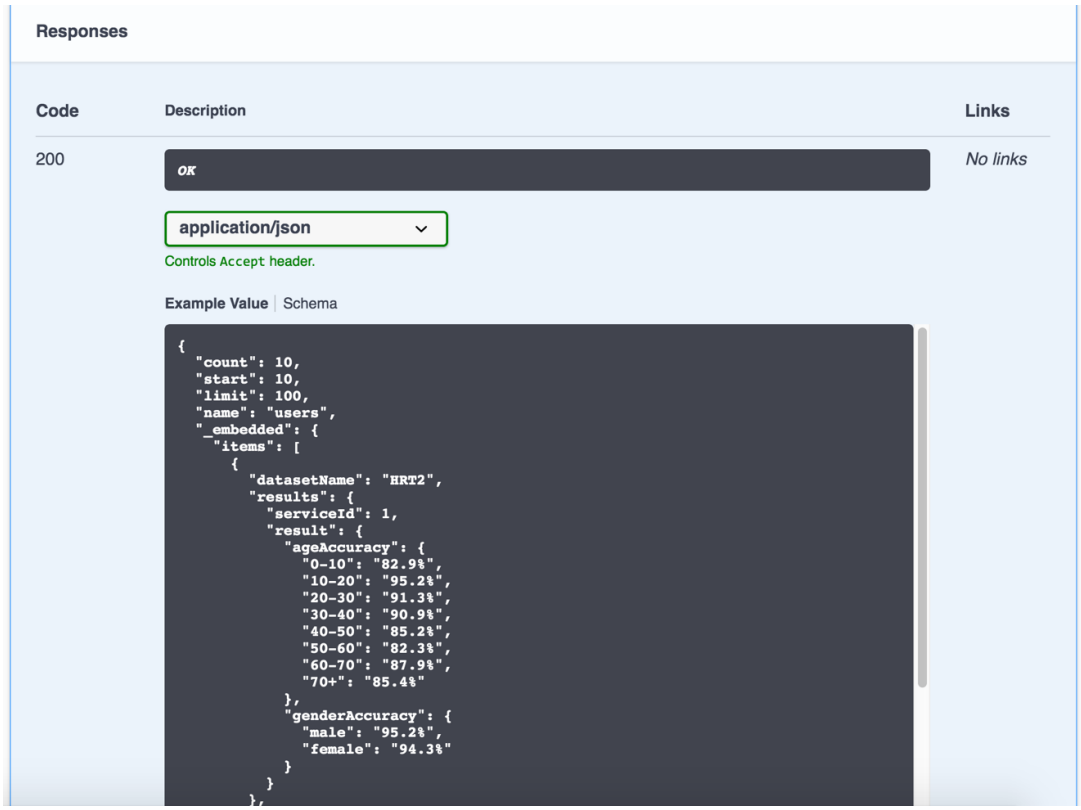

Bench

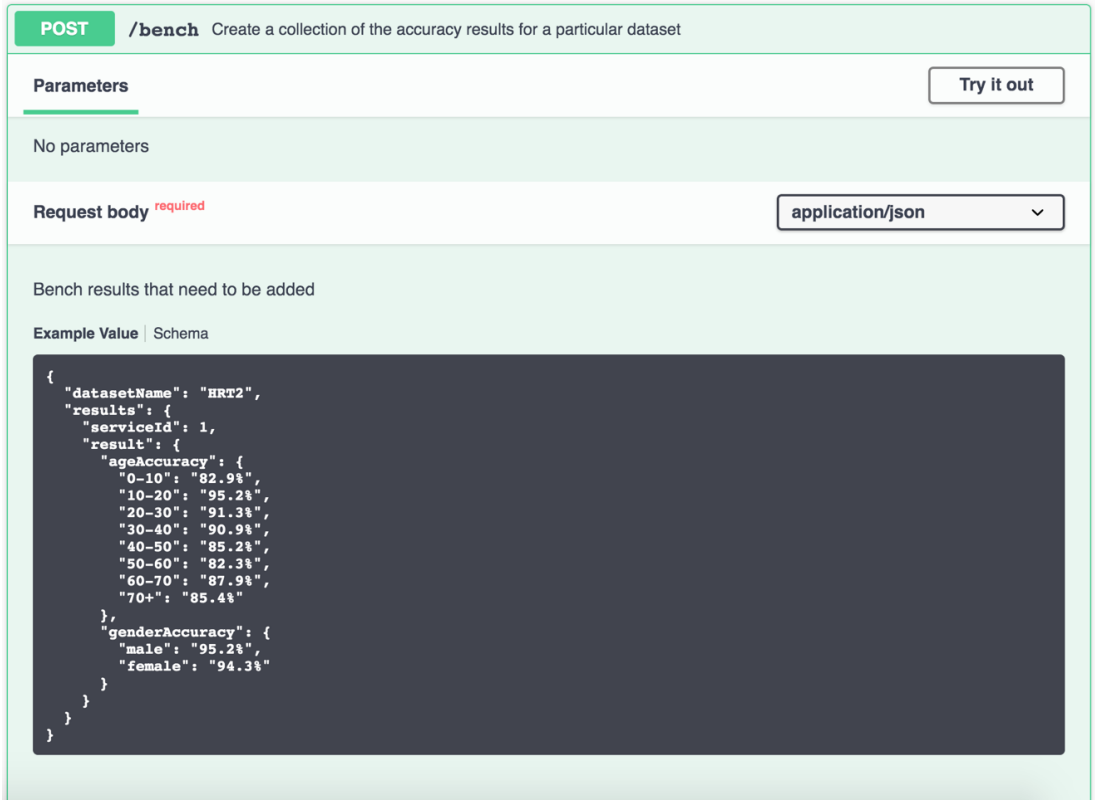
This endpoint manages the benchmarking of services. Put differently, this endpoint fetches and stores metrics about face processing services accuracy in predicting different classes of information, including gender, age, ethnicity, and more. Only the admin of the API can make `POST` or `PATCH` calls to this endpoint, while the end user makes `GET` calls to inquire about which service will currently give the best results for a given subject class (i.e. asian female of age 25 or european male of age 42, etc.) This endpoint will also provide a way of tracking a services progress in improving their algorithms over time. On the front end side, a developer may use this information to provide a visualization for how each service improves over time.

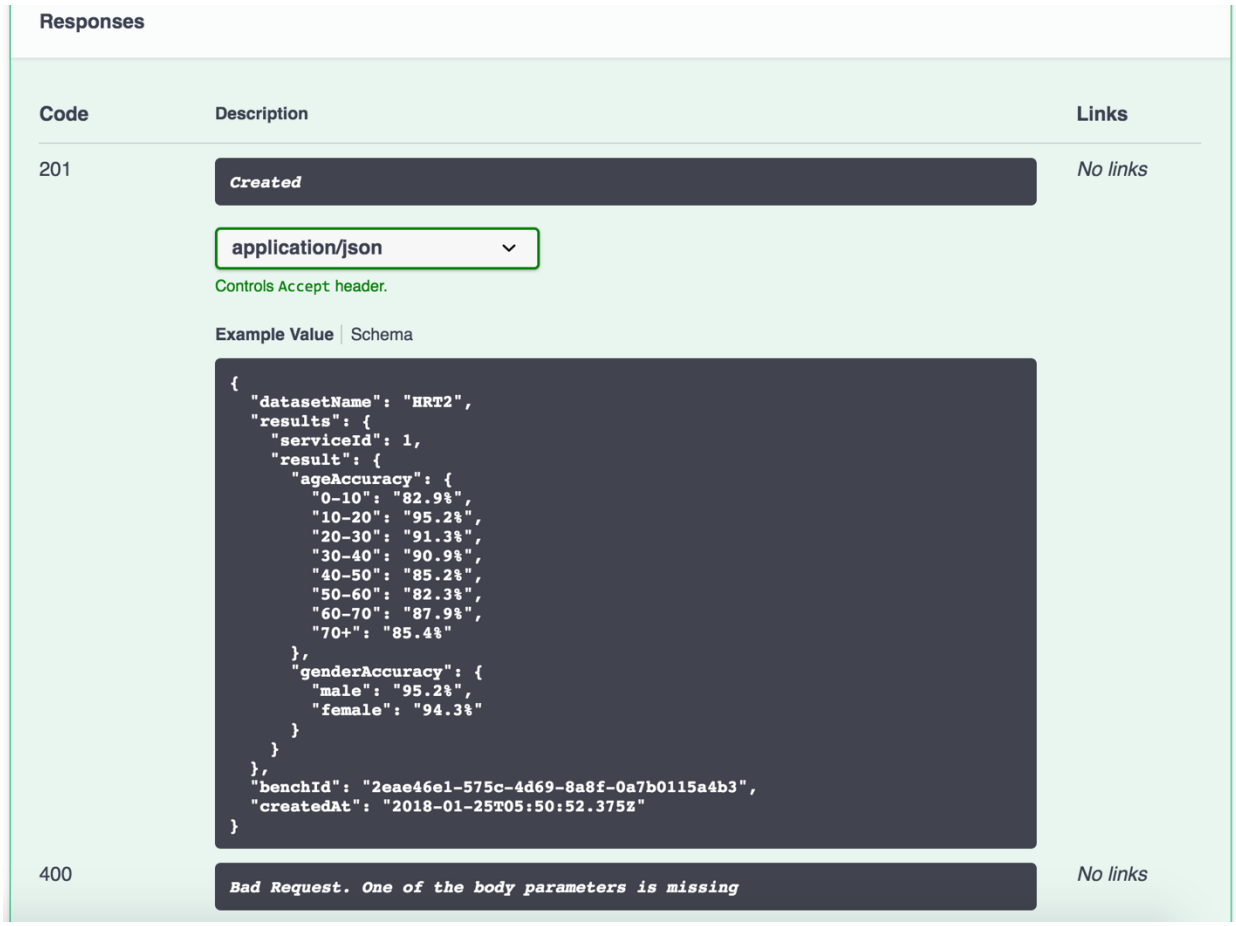
GET `/bench` Return a collection of the accuracy results for all available services for each dataset

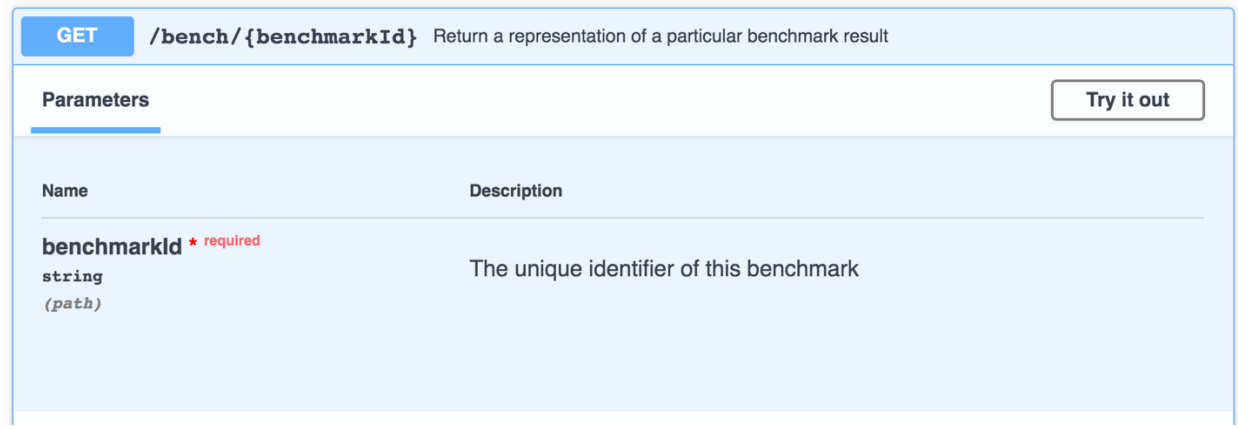
Parameters | Try it out

Name	Description
offset **integer** *(query)*	Number of items to skip before returning the results. *Default value* : 0
limit **integer** *(query)*	Maximum number of items to return. *Default value* : 20
sortBy **string** *(query)*	Optional sort criteria. ?sortBy=field1,-field2`.
filter **string** *(query)*	Optional filter criteria


Responses
Code
Description
Links
200
OK
No links
application/json
Controls Accept header.
Example Value | Schema
{
"count": 10,
"start": 10,
"limit": 100,
"name": "users",
"_embedded": {
"items": [
{
"datasetName": "HRT2",
"results": {
"serviceId": 1,
"result": {
"ageAccuracy": {
"0-10": "82.9%",
"10-20": "95.2%",
"20-30": "91.3%",
"30-40": "90.9%",
"40-50": "85.2%",
"50-60": "82.3%",
"60-70": "87.9%",
"70+": "85.4%"
},
"genderAccuracy": {
"male": "95.2%",
"female": "94.3%"
}
}
},


POST
/bench
Create a collection of the accuracy results for a particular dataset
Parameters
Try it out
No parameters
Request body
required
application/json
Bench results that need to be added
Example Value | Schema
{
"datasetName": "HRT2",
"results": {
"serviceId": 1,
"result": {
"ageAccuracy": {
"0-10": "82.9%",
"10-20": "95.2%",
"20-30": "91.3%",
"30-40": "90.9%",
"40-50": "85.2%",
"50-60": "82.3%",
"60-70": "87.9%",
"70+": "85.4%"
},
"genderAccuracy": {
"male": "95.2%",
"female": "94.3%"
}
}
}
}
}


Responses
Code
Description
Links
201
Created
No links
application/json
Controls Accept header.
Example Value | Schema
{
"datasetName": "HRT2",
"results": {
"serviceId": 1,
"result": {
"ageAccuracy": {
"0-10": "82.9%",
"10-20": "95.2%",
"20-30": "91.3%",
"30-40": "90.9%",
"40-50": "85.2%",
"50-60": "82.3%",
"60-70": "87.9%",
"70+": "85.4%"
},
"genderAccuracy": {
"male": "95.2%",
"female": "94.3%"
}
}
},
"benchId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
"createdAt": "2018-01-25T05:50:52.375Z"
}
400
Bad Request. One of the body parameters is missing
No links


GET
/bench/{benchmarkId}
Return a representation of a particular benchmark result
Parameters
Try it out
Name
Description
benchmarkId * required
string
(path)
The unique identifier of this benchmark

Responses

Code	Description	Links
200	*OK* application/json Controls Accept header. **Example Value** | Schema	*No links*

```
{
  "datasetName": "HRT2",
  "results": {
    "serviceId": 1,
    "result": {
      "ageAccuracy": {
        "0-10": "82.9%",
        "10-20": "95.2%",
        "20-30": "91.3%",
        "30-40": "90.9%",
        "40-50": "85.2%",
        "50-60": "82.3%",
        "60-70": "87.9%",
        "70+": "85.4%"
      },
      "genderAccuracy": {
        "male": "95.2%",
        "female": "94.3%"
      }
    }
  },
  "benchId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
  "createdAt": "2018-01-25T05:50:52.375Z"
}
```

404	*Not found. There is no such bench resource at the specified* `{benchmarkId}` *the* `_error` *field in the response will contain details about the request error.* application/json **Example Value** | Schema	*No links*

```
{
  "_error": {
    "_id": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
    "message": "A resource cannot be found with the given userId",
    "statusCode": 404,
    "attributes": {
      "userId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b4"
    },
    "remediation": "Resubmit the operation with an existing userId",
    "occurredAt": "2018-01-25T05:50:52.375Z"
  }
}
```

Service This API manages the available third party face processing services that are offered through Gladiator.

GET `/services` Return a collection of the available face processing services

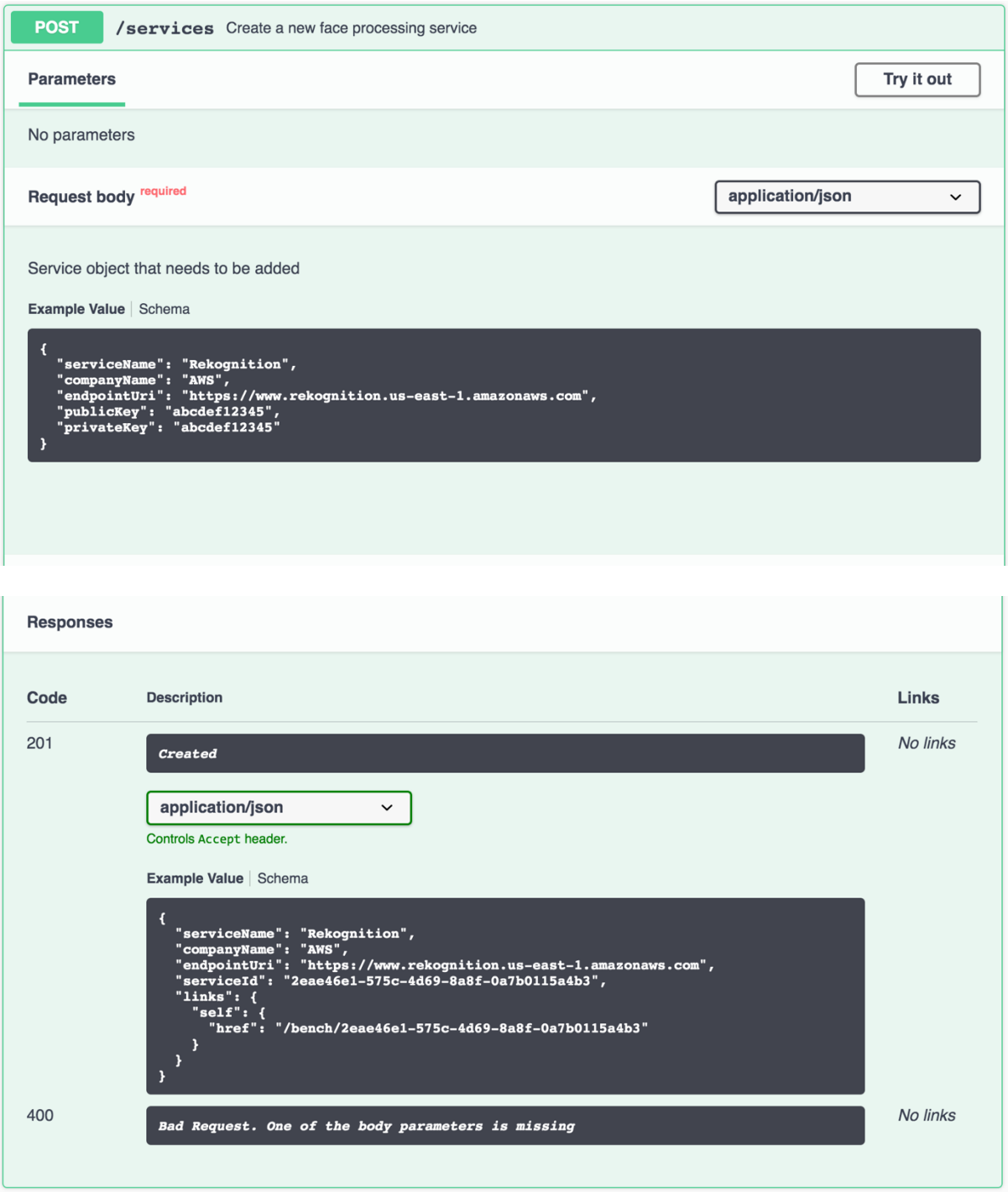
Parameters | **Try it out**

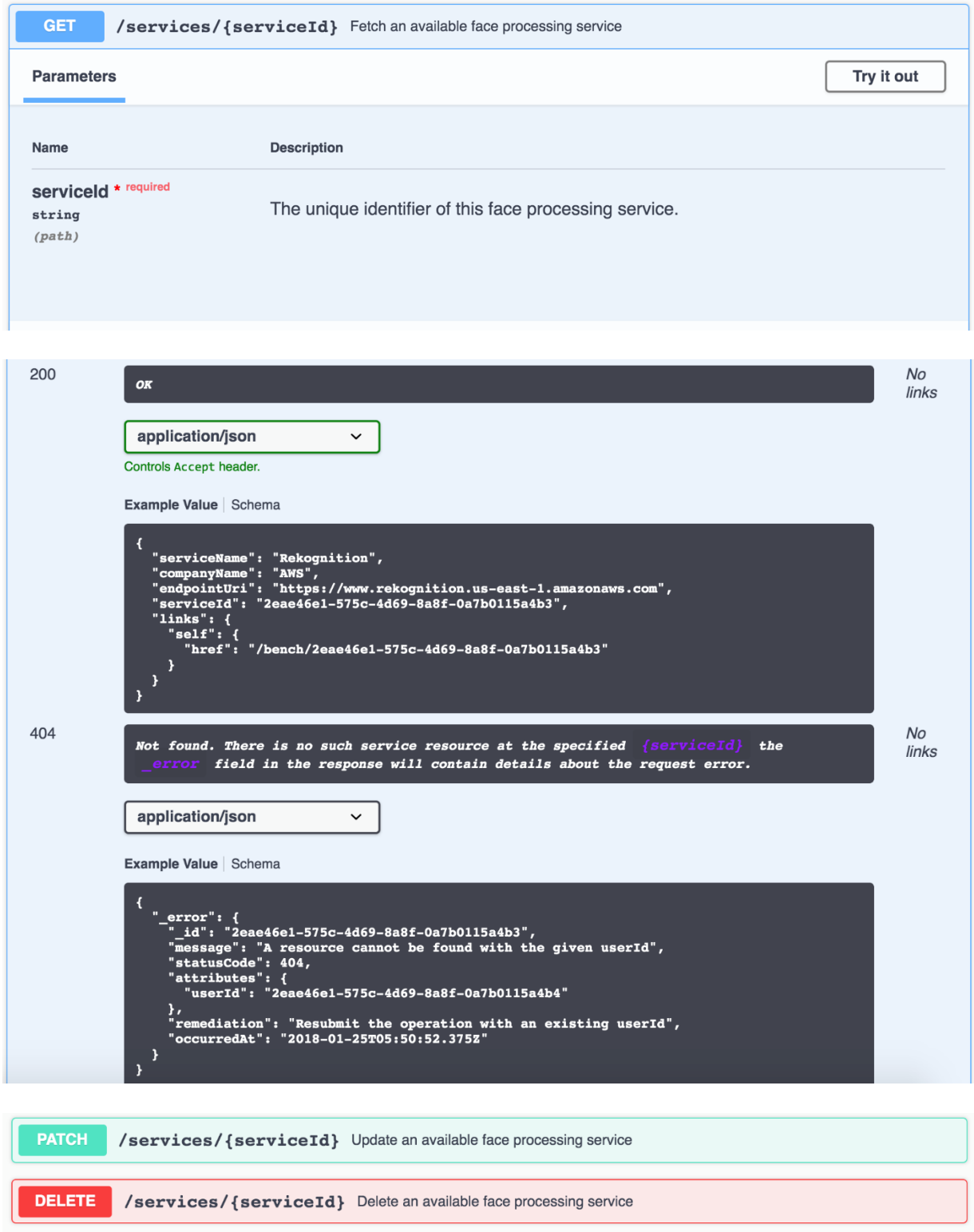
Name	Description
offset **integer** *(query)*	Number of items to skip before returning the results. *Default value* : 0
limit **integer** *(query)*	Maximum number of items to return. *Default value* : 20
sortBy **string** *(query)*	Optional sort criteria. ?sortBy=field1,-field2`.
filter **string** *(query)*	Optional filter criteria

Responses

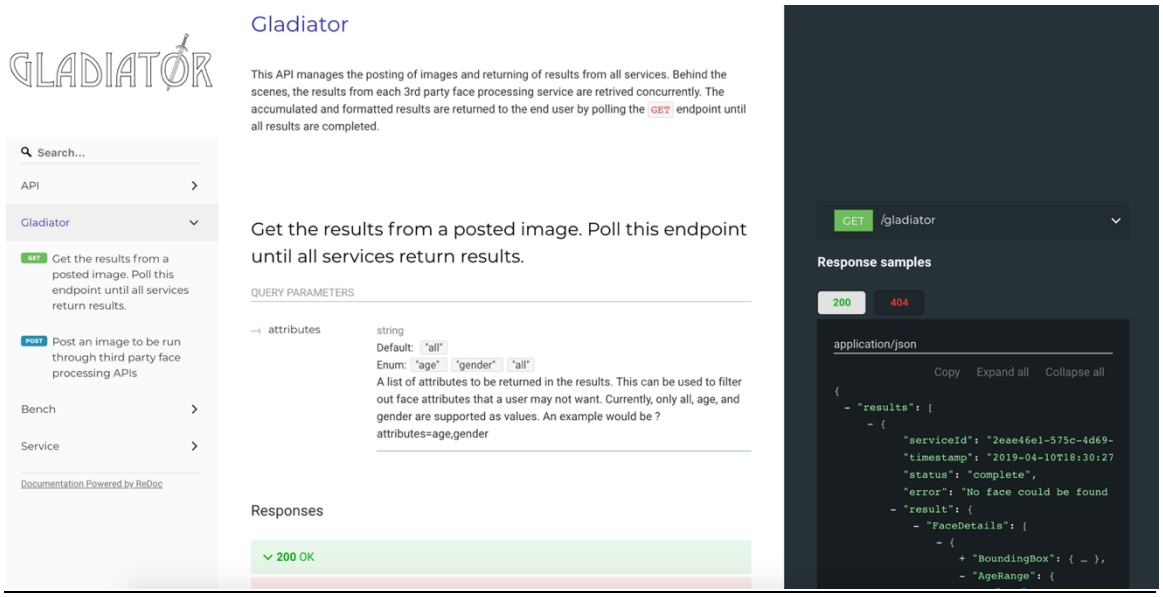
Code	Description	Links
200	*OK* application/json Controls `Accept` header. **Example Value** | Schema	*No links*

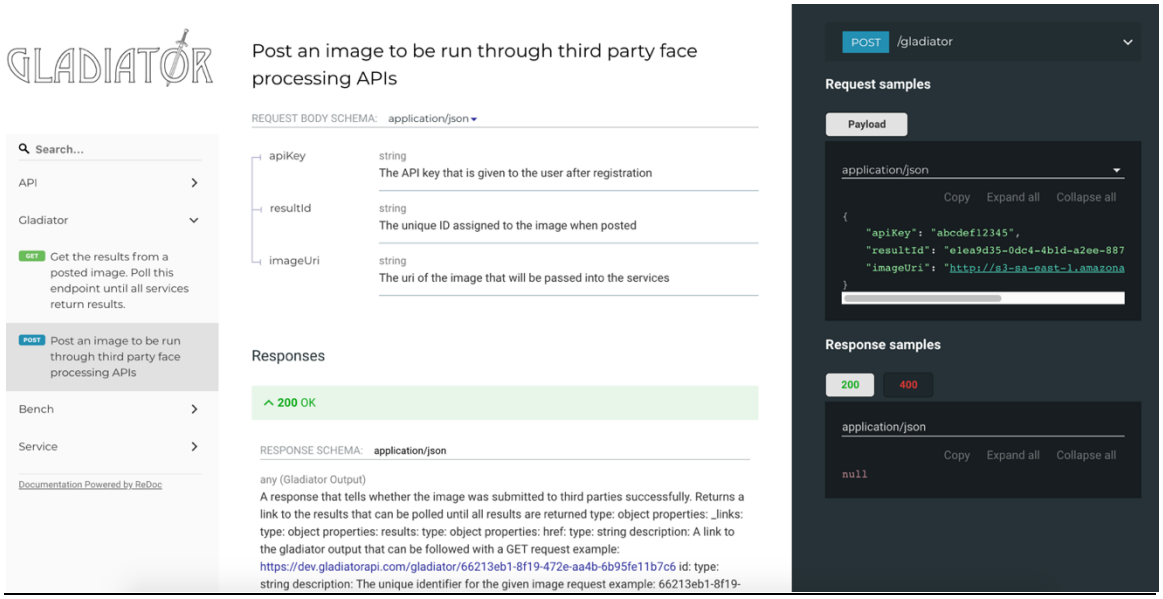

```
{
  "count": 10,
  "start": 10,
  "limit": 100,
  "name": "users",
  "_embedded": {
    "items": [
      {
        "serviceName": "Rekognition",
        "companyName": "AWS",
        "endpointUri": "https://www.rekognition.us-east-1.amazonaws.com",
        "serviceId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
        "links": {
          "self": {
            "href": "/bench/2eae46e1-575c-4d69-8a8f-0a7b0115a4b3"
          }
        }
      }
    ]
  }
}
```

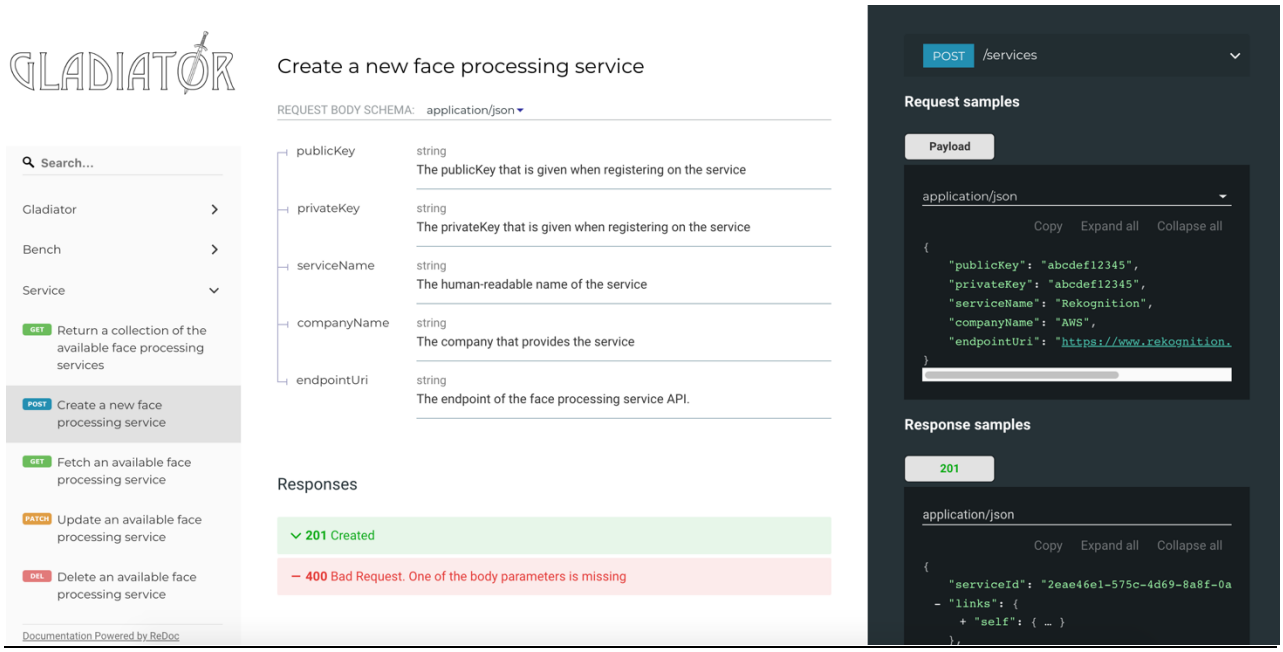

POST
/services
Create a new face processing service
Parameters
Try it out
No parameters
Request body
required
application/json
Service object that needs to be added
Example Value
Schema
{
"serviceName": "Rekognition",
"companyName": "AWS",
"endpointUri": "https://www.rekognition.us-east-1.amazonaws.com",
"publicKey": "abcdef12345",
"privateKey": "abcdef12345"
}
Responses
Code
Description
Links
201
Created
No links
application/json
Controls Accept header.
Example Value
Schema
{
"serviceName": "Rekognition",
"companyName": "AWS",
"endpointUri": "https://www.rekognition.us-east-1.amazonaws.com",
"serviceId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
"links": {
"self": {
"href": "/bench/2eae46e1-575c-4d69-8a8f-0a7b0115a4b3"
}
}
}
400
Bad Request. One of the body parameters is missing
No links

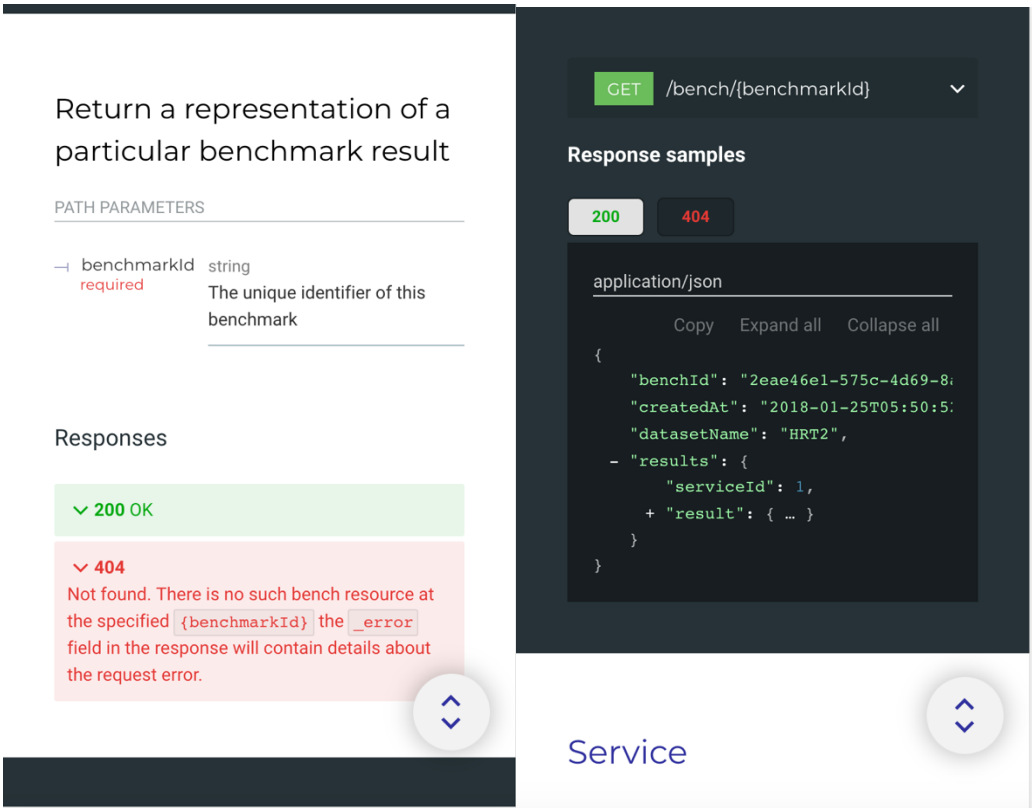

GET
/services/{serviceId}
Fetch an available face processing service
Parameters
Try it out
Name
Description
serviceId * required
string
(path)
The unique identifier of this face processing service.
200
OK
No links
application/json
Controls Accept header.
Example Value | Schema
{
"serviceName": "Rekognition",
"companyName": "AWS",
"endpointUri": "https://www.rekognition.us-east-1.amazonaws.com",
"serviceId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
"links": {
"self": {
"href": "/bench/2eae46e1-575c-4d69-8a8f-0a7b0115a4b3"
}
}
}
404
Not found. There is no such service resource at the specified {serviceId} the _error field in the response will contain details about the request error.
No links
application/json
Example Value | Schema
{
"_error": {
"_id": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b3",
"message": "A resource cannot be found with the given userId",
"statusCode": 404,
"attributes": {
"userId": "2eae46e1-575c-4d69-8a8f-0a7b0115a4b4"
},
"remediation": "Resubmit the operation with an existing userId",
"occurredAt": "2018-01-25T05:50:52.375Z"
}
}
PATCH
/services/{serviceId}
Update an available face processing service
DELETE
/services/{serviceId}
Delete an available face processing service

Appendix B (Example HTML Templates for API Documentation Generated from Swagger)

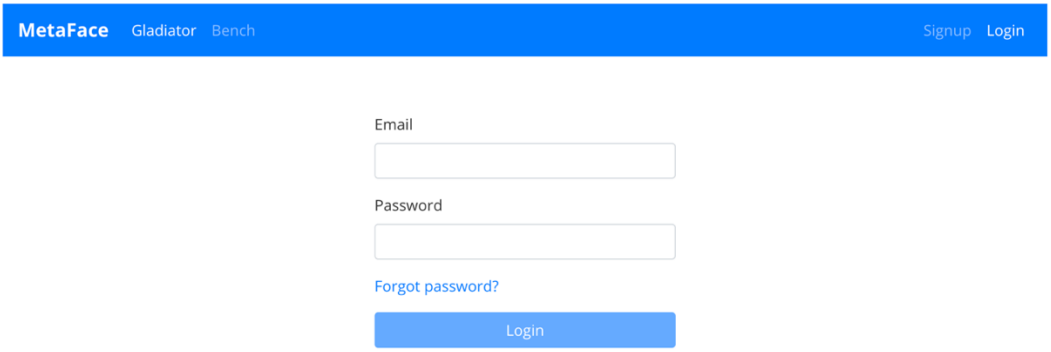




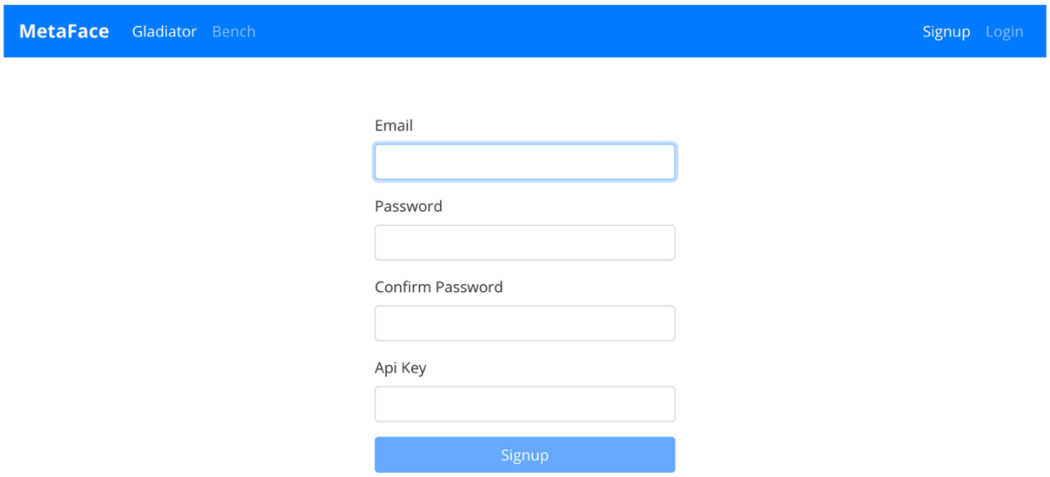

GLADIATOR
Search...
Gladiator
Bench
Service
GET Return a collection of the available face processing services
POST Create a new face processing service
GET Fetch an available face processing service
PATCH Update an available face processing service
DEL Delete an available face processing service
Documentation Powered by ReDoc
Create a new face processing service
REQUEST BODY SCHEMA: application/json
publicKey string The publicKey that is given when registering on the service
privateKey string The privateKey that is given when registering on the service
serviceName string The human-readable name of the service
companyName string The company that provides the service
endpointUri string The endpoint of the face processing service API.
Responses
201 Created
400 Bad Request. One of the body parameters is missing
POST /services
Request samples
Payload
application/json
Copy Expand all Collapse all
"publicKey": "abcdef12345",
"privateKey": "abcdef12345",
"serviceName": "Rekognition",
"companyName": "AWS",
"endpointUri": "https://www.rekognition.
Response samples
201
application/json
Copy Expand all Collapse all
"serviceId": "2eae46e1-575c-4d69-8a8f-0a
"links": {
"self": { … }


Return a representation of a particular benchmark result
PATH PARAMETERS
benchmarkId required string The unique identifier of this benchmark
Responses
200 OK
404
Not found. There is no such bench resource at the specified {benchmarkId} the _error field in the response will contain details about the request error.
GET /bench/{benchmarkId}
Response samples
200
404
application/json
Copy Expand all Collapse all
"benchId": "2eae46e1-575c-4d69-8
"createdAt": "2018-01-25T05:50:5
"datasetName": "HRT2",
"results": {
"serviceId": 1,
"result": { … }
Service

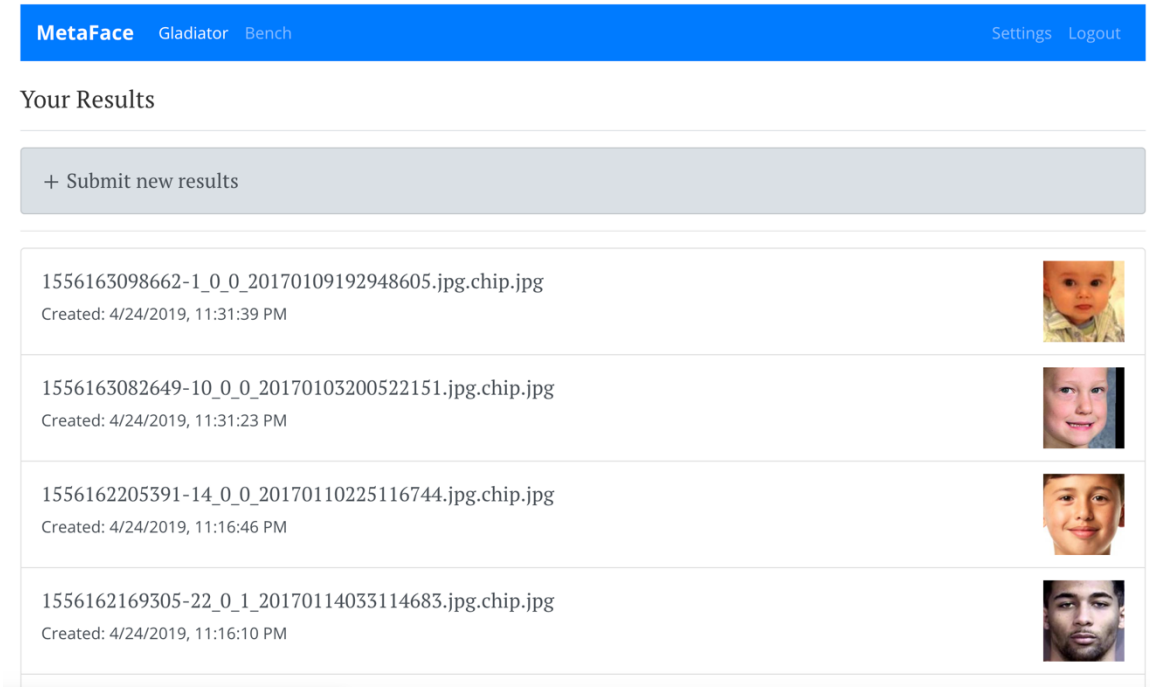
Appendix C (MetaFace Web Application Screenshots)



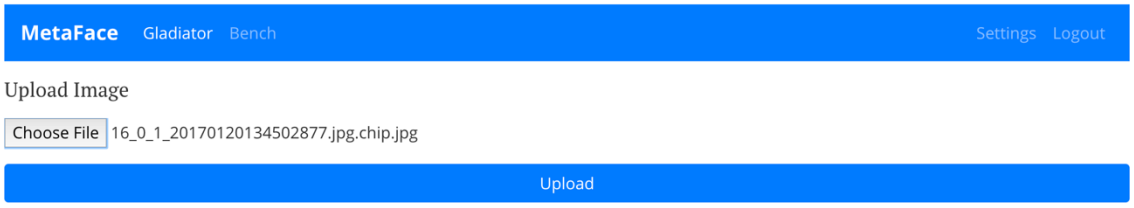
MetaFace login page



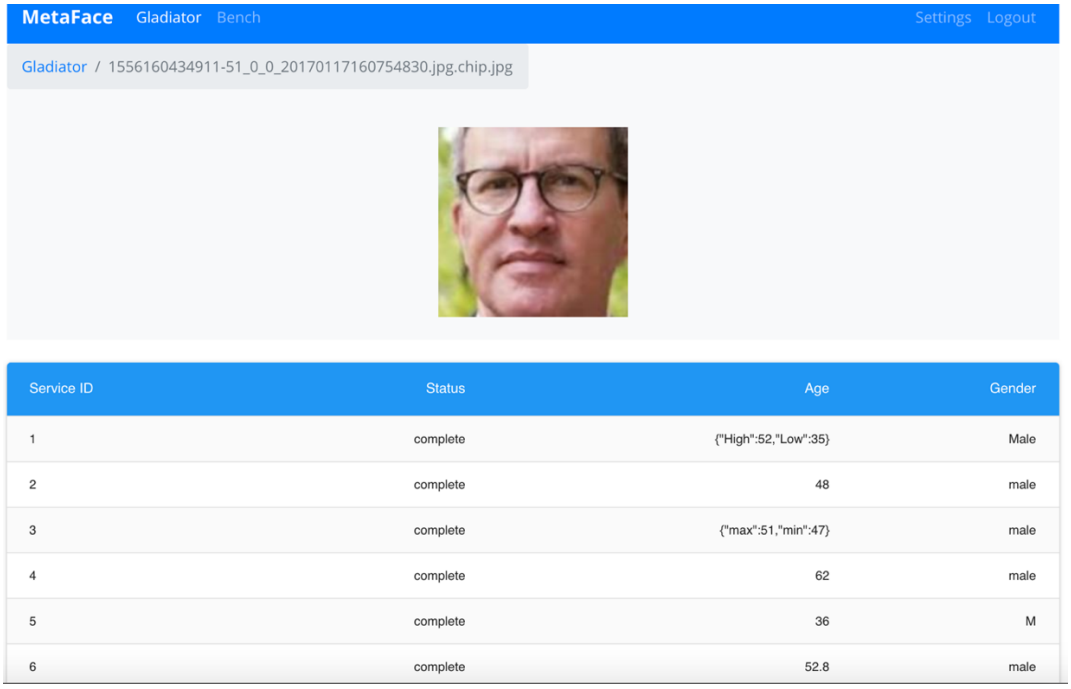
Signup page



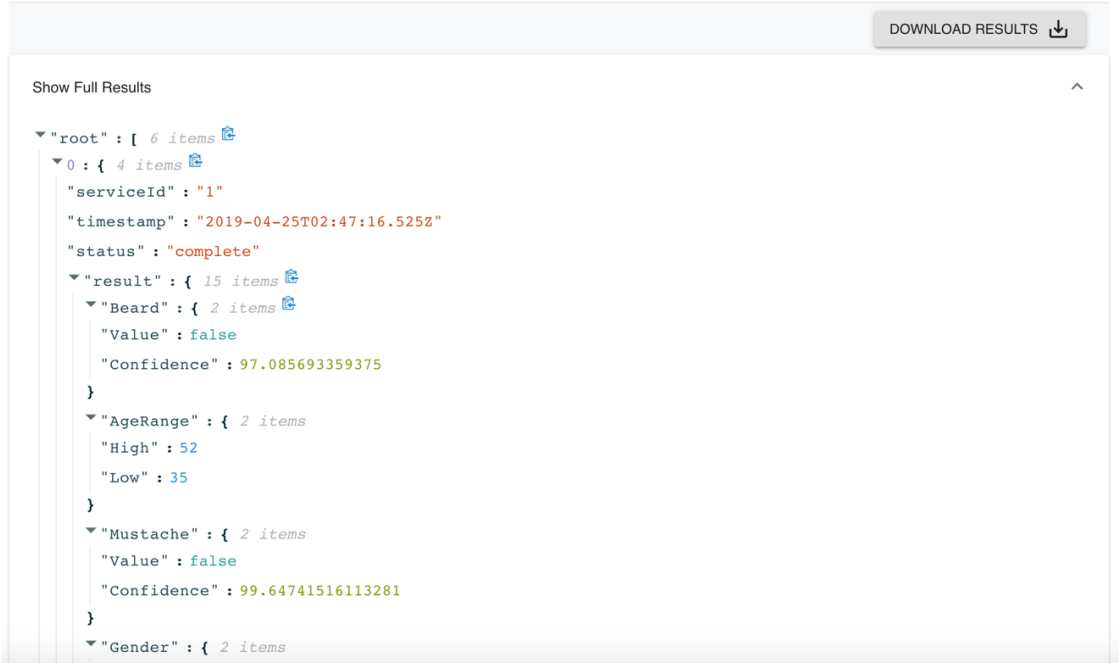
Results list page



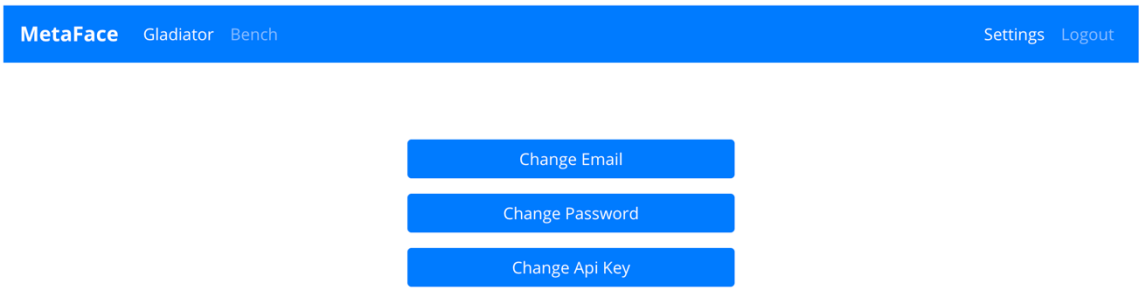
Upload image page



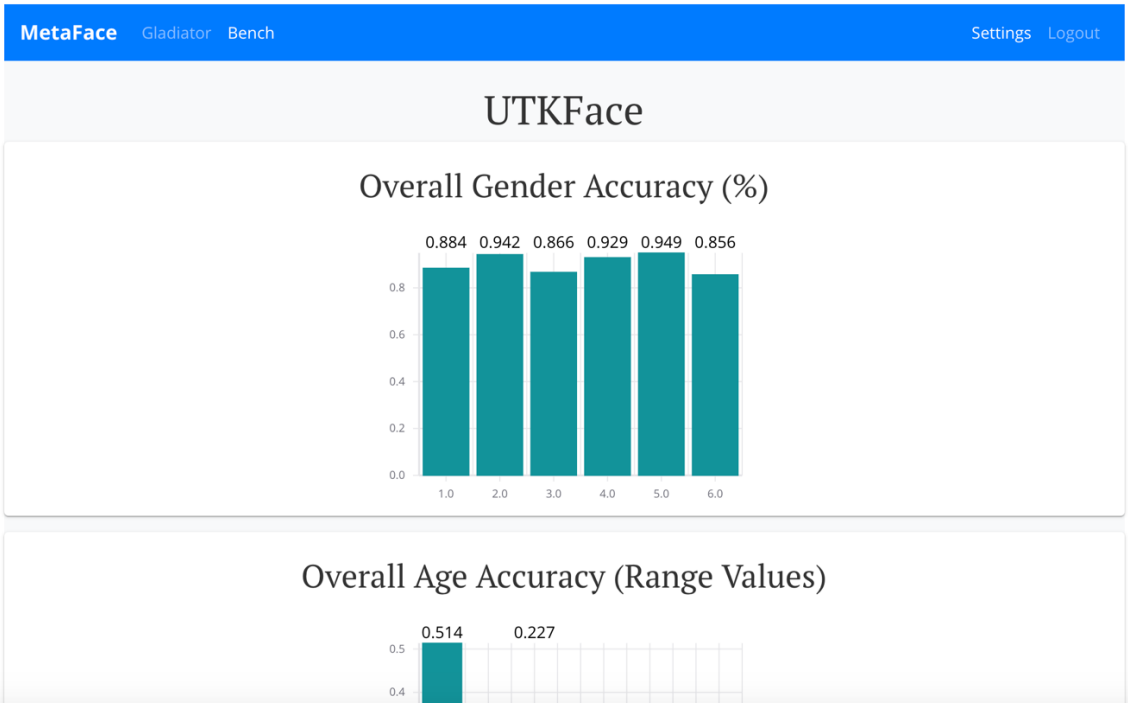
Results page



Results page continued, user clicks dropdown button



User information page



Bench tab

Appendix D (Example of Code Heatmap Generated from Code Coverage)

All files / function getResults.js

92.93% Statements 92/99 **95.45%** Branches 42/44 **100%** Functions 15/15 **92.86%** Lines 91/98

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

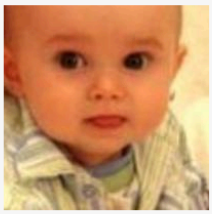
```
1   1x  const serverless = require('serverless-http');
2   1x  const express = require('express');
3   1x  const bodyParser = require('body-parser');
4   1x  const AWS = require('aws-sdk');
5   1x  const { ServiceIds } = require('./util/enums');
6   1x  const { NotFound } = require('../error/notFound');
7
8   1x  const app = express();
9
10  1x  app.use(
11        bodyParser.urlencoded({
12          extended: true,
13        })
14      );
15  1x  app.use(bodyParser.json());
16
17  1x  app.get('/gladiator/:resultId', (req, res) => {
18 12x    const { GLADIATOR_TABLE } = process.env;
19 12x    const dynamoDb = new AWS.DynamoDB.DocumentClient();
20
21 12x    const params = {
22          TableName: GLADIATOR_TABLE,
23          KeyConditionExpression: `resultId = :resultId`,
24          ExpressionAttributeValues: {
25            ':resultId': req.params.resultId,
26          },
27        };
28
29 12x    dynamoDb.query(params, (error, result) => {
30 12x      if (error) {
31  1x        console.log(error);
32  1x        return res.status(500).json({ error: 'Could not get result' });
33        }
```

```
11x     if (result.Item) {
 9x       return res.json(parseResults([result.Item]));
 2x     } else if (result.Items && result.Items.length > 0) {
 1x       return res.json(parseResults(result.Items));
        } else {
 1x       return res
            .status(404)
            .json(
              new NotFound(
                `result with id '${req.params.resultId}' not found`,
                `Resubmit the operation with an existing resultId`
              )
            );
        }
      });
    });

    /**
     * Parse the result set response from dynamoDB and prepare it to return the user.
     * This function assumes that dynamo has returned results from more than 1 serviceId.
     * @param {*} resultSet An array of result sets that is returned from dynamoDB.
     */
 1x const parseResults = function parseResults(resultSet) {
10x   const { resultId, imageUri } = resultSet[0];
10x   const parsedResults = [];

10x   resultSet.forEach(set => {
15x     if (set.serviceId === ServiceIds.rekognition) {
 3x       parsedResults.push(parseRekognition(set));
12x     } else if (set.serviceId === ServiceIds.azure) {
 3x       parsedResults.push(parseAzure(set));
 9x     } else if (set.serviceId === ServiceIds.watson) {
 3x       parsedResults.push(parseWatson(set));
 6x     } else if (set.serviceId === ServiceIds.sighthound) {
 1x       parsedResults.push(parseSighthound(set));
 5x     } else if (set.serviceId === ServiceIds.kairos) {
 2x       parsedResults.push(parseKairos(set));
 3x     } else E if (set.serviceId === ServiceIds.lapetus) {
 3x       parsedResults.push(parseLapetus(set));
      }
    });

```

```
76        // Fill in missing results with pending before timeout
77  15x   const completeServices = parsedResults.map(result => result.serviceId);
78
79  10x   let status = 'pending';
80
81  10x   I if (new Date().toISOString() - parsedResults[0].timestamp > 100 * 30) {
82          status = 'failed';
83        }
84
85  10x   Object.values(ServiceIds).forEach(id => {
86  60x     if (completeServices.indexOf(id) < 0) {
87  45x       parsedResults.push({
88              serviceId: id,
89              status,
90            });
91        }
92      });
93
94  10x   return {
95          resultId,
96          imageUri,
97  63x     results: parsedResults.sort((a, b) => a.serviceId > b.serviceId),
98        };
99      };
100
101 1x  const parseResult = function parseResult(response, result) {
102       // Todo: handle multiple faces!
103 8x    return {
104         serviceId: response.serviceId,
105         timestamp: response.timestamp,
106         status: 'complete',
107         result,
108       };
109     };
110
111 1x  const errorResponse = function errorResponse(result, err) {
112 7x    return {
113         serviceId: result.serviceId,
114         timestamp: result.timestamp,
115         status: 'failed',
116         error: err,
117       };
118     };
119
```

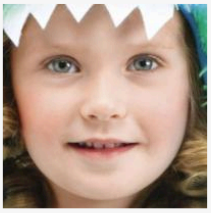
Appendix E (Example Results from Gladiator API displayed on MetaFace Web Application)

Gladiator / 1556163098662-1_0_0_20170109192948605.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":5,"Low":1}	Female
2	complete	0	female
3	complete	{"max":12,"min":0}	female
4	complete	0	male
5	complete	31	F
6	complete	20.5	male

Gladiator / 1556426485162-7_1_0_20170109192242639.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":9,"Low":4}	Female
2	complete	5	female
3	complete	{"max":12,"min":0}	female
4	complete	1	female
5	complete	26	F
6	complete	25.6	female

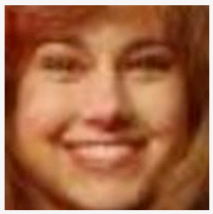
Gladiator / 1556162205391-14_0_0_20170110225116744.jpg.chip.jpg

Service ID	Status	Age	Gender
1	complete	{"High":18,"Low":11}	Male
2	complete	7	male
3	complete	{"max":12,"min":0}	female
4	complete	10	male
5	complete	35	M
6	complete	18.6	male

Gladiator / 1556426118788-19_1_0_20170103201503695.jpg.chip.jpg

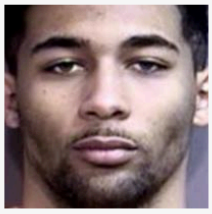
Service ID	Status	Age	Gender
1	complete	{"High":38,"Low":20}	Female
2	complete	24	female
3	complete	{"max":26,"min":23}	female
4	complete	27	female
5	complete	32	F
6	complete	31.7	male

Gladiator / 1556425916354-20_1_0_20170120140127200.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":43,"Low":26}	Female
2	complete	22	female
3	failed	-	-
4	complete	21	female
5	complete	37	F
6	failed	-	-

Gladiator / 1556162169305-22_0_1_20170114033114683.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":38,"Low":23}	Male
2	complete	25	male
3	complete	{"max":21,"min":19}	male
4	complete	29	male
5	complete	29	M
6	complete	27.4	male

Gladiator / 1556161928058-23_0_0_20170116221708213.jpg.chip.jpg

Service ID	Status	Age	Gender
1	complete	{"High":38,"Low":20}	Male
2	complete	25	male
3	complete	{"max":21,"min":19}	male
4	complete	69	male
5	complete	31	M
6	complete	27.1	male

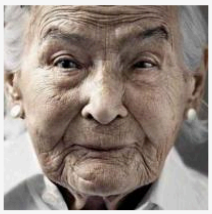
Gladiator / 1556161779065-30_0_0_20170117143352248.jpg.chip.jpg

Service ID	Status	Age	Gender
1	complete	{"High":38,"Low":23}	Male
2	complete	27	male
3	complete	{"max":28,"min":25}	male
4	complete	25	male
5	complete	33	M
6	complete	28.8	male

Gladiator / 1556160434911-51_0_0_20170117160754830.jpg.chip.jpg

Service ID	Status	Age	Gender
1	complete	{"High":52,"Low":35}	Male
2	complete	48	male
3	complete	{"max":51,"min":47}	male
4	complete	62	male
5	complete	36	M
6	complete	52.8	male

Gladiator / 1556160270691-100_1_2_20170112213615815.jpg.chip.jpg



Service ID	Status	Age	Gender
1	complete	{"High":90,"Low":60}	Female
2	complete	71	male
3	complete	{"max":76,"min":73}	male
4	complete	95	female
5	complete	53	F
6	complete	79.6	male

REFERENCES

[1] "JavaScript." *MDN Web Docs*, Mozilla, developer.mozilla.org/en-US/docs/Glossary/JavaScript.

[2] Freeman, Eric, and Elisabeth Robson. *Head First JavaScript Programming*. OReilly, 2014.

[3] Subramaniam, Venkat. *Rediscovering JavaScript Master ES6, ES7, and ES8*. The Pragmatic Bookshelf, 2018.

[4] Osmani, Addy. *Learning JavaScript Design Patterns*. 3rd ed., OReilly Media, 2013.

[5] "Top Languages Over Time." *The State of the Octoverse*, GitHub, 2018, octoverse.github.com/projects#languages.

[6] "TIOBE Index for March 2019." *TIOBE*, Mar. 2019, www.tiobe.com/tiobe-index/.

[7] Webber, Adam Brooks. *Modern Programming Languages: A Practical Introduction*. 2nd ed., Franklin, Beedle & Associates, 2013.

[8] Alicea, Anthony. "Learn and Understand NodeJS." *Udemy*, 2018, www.udemy.com/understand-nodejs.

[9] Lebec, Gabriel. "JavaScript: A History for Beginners." *Course Report*, 12 Mar. 2019, www.coursereport.com/blog/history-of-javascript.

[10] Toledo, Judi. "Why Millions of Developers Use JavaScript for Web Application Development?" *Torque*, Torque, 29 May 2018, torquemag.io/2018/06/why-millions-of-developers-use-javascript-for-web-application-development/.

[11] Nemeth, Gergely. "History of Node.js on a Timeline." *RisingStack*, 21 Mar. 2018, blog.risingstack.com/history-of-node-js/.

[12] "NPM." *About Npm | Npm Documentation*, NPM, docs.npmjs.com/about-npm/.

[13] “AWS Documentation.” *Amazon*, Amazon, docs.aws.amazon.com/

[14] Baron, Joe, et al. *AWS Certified Solutions Architect Official Study Guide: Associate Exam*. Sybex, a Wiley Brand, 2017.

[15] Kroonenburg, Ryan. *A Cloud Guru*, A Cloud Guru, acloud.guru/learn/aws-certified-solutions-architect-associate.

[16] “What Is YAML? A Beginner's Guide.” *CircleCI*, 12 Apr. 2017, circleci.com/blog/what-is-yaml-a-beginner-s-guide/.

[17] “Introducing JSON.” *JSON*, www.json.org/.

[18] “Extensible Markup Language (XML).” *W3C*, www.w3.org/XML/.

[19] “Cognito User Pool vs Identity Pool.” *Serverless Stack*, 5 Jan. 2017, serverless-stack.com/chapters/cognito-user-pool-vs-identity-pool.html.

[20] Hahn, Evan. *Express in Action: Writing, Building, and Testing Node.js Applications*. Manning, 2016.

[21] “Npm.” *Npm*, www.npmjs.com/.

[22] “Jasmine Documentation.” *Jasmine*, jasmine.github.io/.

[23] “Automate and Enhance Your Workflow.” *Gulp.js - The Streaming Build System*, gulpjs.com/.

[24] “Prettier · Opinionated Code Formatter.” *Prettier*, prettier.io/.

[25] “Pluggable JavaScript Linter.” *ESLint*, eslint.org/.

[26] Dwyl. “Dwyl/Aws-Sdk-Mock.” *GitHub*, 8 Apr. 2019, github.com/dwyl/aws-sdk-mock.

[27] Visionmedia. “Visionmedia/Supertest.” *GitHub*, 15 Mar. 2019, github.com/visionmedia/supertest.

[28] Istanbuljs. “Istanbuljs/Nyc.” *GitHub*, 10 Apr. 2019, github.com/istanbuljs/nyc.

[29] “Eslint-Config-Airbnb.” *Npm*, www.npmjs.com/package/eslint-config-airbnb.

[30] “Chai Assertion Library.” *Chai*, www.chaijs.com/.

[31] “RightScale 2019 State of the Cloud Report.” *RightScale*, Flexera, 2019, www.rightscale.com/lp/state-of-the-cloud.

[32] Watson-Developer-Cloud. “Watson-Developer-Cloud/Node-Sdk.” *GitHub*, IBM, 12 Apr. 2019, github.com/watson-developer-cloud/node-sdk.

[33] “Class: AWS.Rekognition.” *Amazon*, Amazon, docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Rekognition.html.

[34] Body-Parser. “Expressjs/Body-Parser.” *GitHub*, Express, 8 Aug. 2018, github.com/expressjs/body-parser.

[35] Biehl, Matthias. *RESTful Web API Design: APIs Your Consumers Will Love*. API-University Press, 2016.

[36] Levin, Guy. “REST API Error Handling Best Practices.” *REST API and Beyond*, REST API and Beyond, 14 Sept. 2016, blog.restcase.com/rest-api-error-codes-101/.

[37] “The Serverless Application Framework Powered by AWS Lambda, API Gateway, and More.” *Serverless*, serverless.com/.

[38] Serverless-Http. “Dougmoscrop/Serverless-Http.” *GitHub*, 14 Apr. 2019, github.com/dougmoscrop/serverless-http.

[39] "What Is Serverless Architecture?" *Twilio*, www.twilio.com/docs/glossary/what-is-serverless-architecture.

[40] Serverless-domain-manager. "Amplify-Education/Serverless-Domain-Manager." *GitHub*, 22 Mar. 2019, github.com/amplify-education/serverless-domain-manager.

[41] "Serverless Computing – Amazon Web Services." *Amazon*, Amazon, aws.amazon.com/serverless/.

[42] "Postman." *Postman*, www.getpostman.com/.

[43] Brody, Alon. "SQL vs NoSQL: The Differences Explained." *SQL vs NoSQL: The Differences Explained*, Panoply, 29 Nov. 2018, blog.panoply.io/sql-or-nosql-that-is-the-question.

[44] "What Are APIs?" *Red Hat - We Make Open Source Technologies for the Enterprise*, www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

[45] Virdee-Chapman, Ben. "Face Recognition: Kairos vs Microsoft vs Google vs Amazon vs OpenCV." *Kairos*, 9 Jan. 2017, www.kairos.com/blog/face-recognition-kairos-vs-microsoft-vs-google-vs-amazon-vs-opencv.

[46] Bobriakov, Igor. "Comparison of the Top Cloud APIs for Computer Vision." *Data Science Central*, www.datasciencecentral.com/profiles/blogs/comparison-of-the-top-cloud-apis-for-computer-vision.

[47] Walling, Alex, and Alex WallingDeveloper. “Top 10 Facial Recognition APIs (Updated for 2019) | RapidAPI.” *RapidAPI Blog*, 12 Apr. 2019, blog.rapidapi.com/top-facial-recognition-apis/.

[48] “Amazon Rekognition – Video and Image - AWS.” *Amazon*, Amazon, aws.amazon.com/rekognition/.

[49] “Face API - Facial Recognition Software | Microsoft Azure.” *API - Facial Recognition Software | Microsoft Azure*, azure.microsoft.com/en-us/services/cognitive-services/face/.

[50] “Visual Recognition.” *Watson Visual Recognition*, 28 Nov. 2016, www.ibm.com/watson/services/visual-recognition/.

[51] “Sighthound.io.” *Sighthound, Inc.*, www.sighthound.com/products/sighthound-io.

[52] Belyeu, Rajnesah. “Serving Businesses with Face Recognition.” *Kairos*, www.kairos.com/.

[53] “Lapetus Solutions, Inc.” *Lapetus Solutions, Inc.*, www.lapetussolutions.com/.

[54] *UTKFace*, susanqq.github.io/UTKFace/.

[55] Chinnathambi, Kirupa. *Learning React: a Hands-on Guide to Building Maintainable, High-Performing Web Application User Interfaces Using the React Jav*. 2nd ed., Addison-Wesley, 2016.

[56] “React – A JavaScript Library for Building User Interfaces.” *– A JavaScript Library for Building User Interfaces*, reactjs.org/.

[57] “The World's Most Popular React UI Framework - Material-UI.” *Material*, material-ui.com/.

[58] Otto, Mark, and Jacob Thornton. “Bootstrap.” · *The Most Popular HTML, CSS, and JS Library in the World.*, getbootstrap.com/.

[59] “React Bootstrap.” *React*, react-bootstrap.github.io/.

[60] “Facial Recognition's 'Dirty Little Secret': Social Media Photos Used without Consent.” *NBCNews.com*, NBCUniversal News Group, www.nbcnews.com/tech/internet/facial-recognition-s-dirty-little-secret-millions-online-photos-scraped-n981921.

[61] Connellan, Shannon, and Shannon Connellan. “Your Social Media Photos Could Be Training Facial Recognition AI without Your Consent.” *Mashable*, Mashable, 13 Mar. 2019, mashable.com/article/ibm-flickr-images-training-facial-recognition-system/.

[62] Buolamwini, Joy. “Facial Recognition Technology Is Both Biased and Understudied – MIT Media Lab.” *MIT Media Lab*, www.media.mit.edu/articles/facial-recognition-technology-is-both-biased-and-understudied/.

[63] “21 Amazing Uses for Face Recognition – Facial Recognition Use Cases.” *FaceFirst Face Recognition Software*, 14 Mar. 2019, www.facefirst.com/blog/amazing-uses-for-face-recognition-facial-recognition-use-cases/.

[64] ReportBuyer. “The Global Facial Recognition Market Is Expected to Grow from $4.05 Billion in 2017 to Reach $14.95 Billion by 2026 with a CAGR of 15.6%.” *PR Newswire: Press Release Distribution, Targeting, Monitoring and Marketing*, 30 Aug. 2018, www.prnewswire.com/news-releases/the-global-facial-

recognition-market-is-expected-to-grow-from-4-05-billion-in-2017-to-reach-14-95-billion-by-2026-with-a-cagr-of-15-6-300704785.html.

[65] Wodehouse, Carey. “Intro to APIs: What Are They & What Do They Do?” *Upwork*, 17 Oct. 2018, www.upwork.com/hiring/development/intro-to-apis-what-is-an-api/.