

B. TECH. PROJECT REPORT

On

Web Development - Creating a Serverless Microservice to extract Human Names from screen

BY
Rajat Upadhyay



**DISCIPLINE OF METALLURGY ENGINEERING AND MATERIALS
SCIENCE**
INDIAN INSTITUTE OF TECHNOLOGY INDORE
December 2019

Web Development - Creating a Serverless Microservice to extract Human Names from screen

A PROJECT REPORT

*Submitted in partial fulfillment of the
requirements for the award of the degrees
of*

BACHELOR OF TECHNOLOGY

in

METALLURGY ENGINEERING AND MATERIALS SCIENCE

Submitted by:

Rajat Upadhyay

Guided by:

**Dr. Vinod Kumar, Assistant Professor, Department of Metallurgy Engineering and
Materials Science, IIT Indore**

Mr. Ankit Maheshwari, Vice President, Engineering, Innovaccer Analytics Pvt. Ltd.




INDIAN INSTITUTE OF TECHNOLOGY INDORE

December 2019

CANDIDATE'S DECLARATION

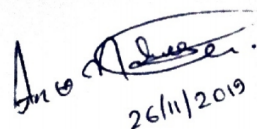
I hereby declare that the project entitled “**Web Development - Creating a serverless microservice to extract Human names from screen**” submitted in partial fulfillment for the award of the degree of Bachelor of Technology in ‘Metallurgy Engineering and Materials Science’ completed under the supervision of **Dr. Vinod Kumar (Assistant Professor, Department of Metallurgy Engineering and Materials Science, IIT Indore)** and **Mr. Ankit Maheshwari (Vice President, Engineering, Innovaccer Analytics Pvt. Ltd.)** is an authentic work.

Further, I declare that I have not submitted this work for the award of any other degree elsewhere.


Rajat Upadhyay 26/11/2019
Signature and name of the student(s) with date

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.


26/11/2019 [VP, Engineering]

Signature of BTP Guide(s) with dates and their designation

Preface

This report on “Web Development - Creating a Serverless Microservice to extract Human Names from Screen ” is prepared under the guidance of Dr. Vinod Kumar, IIT Indore and Mr. Ankit Maheshwari, Innovaccer Analytics Pvt. Ltd.

Through this report I have tried to give a detailed procedure to build a Serverless Microservice that works in conjunction with a configuration driven backend to extract Patient names from screen and display the medical information of the patient to optimise the Patient Provider Engagement.

Rajat Upadhyay

160005028

B.Tech. IV Year

Discipline of Metallurgy Engineering and Materials Science

IIT Indore

Acknowledgements

I wish to thank Dr. Vinod Kumar and Mr. Ankit Maheshwari for their kind support and valuable guidance.

It is their help and support, due to which I was able to complete the design and technical report. Without their support this report would not have been possible.

I wish to thank them for giving me this opportunity to work on this project which has tremendous future scope and which provided me the chance to learn about such a unique and efficient way to develop applications. This knowledge will be very helpful for my career.

Rajat Upadhyay

160005028

B.Tech. IV Year

Discipline of Metallurgy Engineering and Materials Science

IIT Indore

List Of Figures

Figure No.	Page No.
1. Sample EMR	2
2. Docker Engine	9
3. Docker Architecture	11
4. EC2 can be accessed through SSH	13
5. Structure of a Django Project	15
6. MVC Pattern	16
7. MVT Architecture in Django	17
8. Architecture of Django	18
9. Features of MongoDB	19
10. Comparison Between RDBMS and MongoDB	21
11. Complete system for Image Processing	22
12. Division Plan	22
13. Digitization	23
14. Digital Matrix	23
15. Three phases of pattern recognition	24
16. Pixel Grid	25
17. Explaining Layers in Lambda.	28
18. How AMI is launched	29
19. How Instances are run	30
20. Deploying Lambda with S3 and API Gateway, using Serverless	31
21. Function to determine the name of the active window on screen	33
22. Function to determine number of monitors and grab screenshot	33
23. Function to normalise Coordinates of the application window	35
24. Code to grab and save screenshot of the application window	36
25. Extracting text from Image	36
26. Using regex to detect patterns in the text	37
27. Configuration in MongoDB which determines the behaviour of the code	38
28. CI/CD integration with Jenkins explained	39

Chapters:

Title	Page No.
1. Introduction: - - - - -	2-4:
i. EMRs	2
ii. Servers	3
iii. Name Parsing	3
iv. Serverless Architecture	3
v. Problem Statement	4
2. Literature: - - - - -	5-25
i. AWS Lambda	6
ii. Docker	8
iii. EC2 AMI	12
iv. Django Framework	13
v. MongoDB	19
vi. Image Processing	21
3. Methodology/Experiment: - - - - -	26-41
a. Approach, Difficulties and Solutions:	
i. Serverless Microservice	26
ii. Text extraction and name parsing	37
iii. Configuration driven backend	37
iv. Continuous Integration and Deployment	39
4. Results and Discussions - - - - -	43
5. Future Scope - - - - -	44
6. References - - - - -	45

Chapter - 1

Introduction:

1.1 EMR:

EMR stands for Electronic medical records, which are digital equivalent of paper records, or charts at a clinician's office. EMRs typically contain general information such as treatment and medical history of a patient as it is collected by the individual medical practice. By implementing EMR, patient data can be tracked over an extended period of time by multiple healthcare providers. It can help identify those who are due for preventive checkups and screenings and monitor how each patient measures up to certain requirements like vaccinations and blood pressure readings. EMRs are designed to help organisations provide efficient and precise care. Perhaps the most significant difference is that EMR record are universal, meaning that instead of having different charts at different healthcare facilities, a patient will have one electronic chart that can be accessed from any healthcare facility using EMR software.



Fig.1 : Sample EMR

1.2 Servers:

All the applications and websites have some code/program running behind the curtains to provide the user Interface and services. To run and compute, the program needs a processor and memory. Without the memory, the code/program cannot be executed. When an application is run in desktops/laptops, the application uses some of the primary memory of the system to run the programs written to make the application. This memory is called primary memory (RAM). In case of websites, the code written in the website cannot use the computer's memory to get executed and return the result. This computing is done on Servers. A server is a computer device that provides a service to another computer or its user, also known as client. It is basically a computing machine that executes the code deployed on it when requested, and returns the computed result as response. The code behind all the websites and a lot of applications are hosted on servers. Some of the leading companies that offer server computing are: Amazon(AWS), Google(GCP), Microsoft(Azure), IBM , Salesforce etc. These companies offer servers on which applications can be deployed by paying the price for hosting the website/application. This is one of the major sources of income for these companies.

1.3 Human Name Parsing:

Giants like Google, Amazon, Microsoft dominate the tech world. Apart from providing services to people all over the world they also act as the backbone of most of the websites and applications by providing Cloud computing service(servers) to host and deploy the code. But even these tech giants fail to solve the problem of human name parsing. They do provide service to help recognise Human names, but they fail to deliver high accuracy. This problem can be dealt with by using these services along with Python, Regular Expressions and Image Processing.

1.4 Serverless Architecture and Configuration driven Backend:

One of the major problems faced after making and deploying any website is maintenance and upgrades. To implement a change as small as a single line of code, the entire website has to re-deployed on the server. To add every feature, to remove every error, to fix every bug the same tiresome process of deployment has to be done. The correct versions of codes have to be merged, unit tests have to run and an integration/deployment pipeline has to be run to re-deploy the code on the server. This can become a major pain point as the application grows and can cause some serious scalability issues. Semi-automated pipelines can help but even they need to be maintained and upgraded with time. To solve this problem, two changes can be implemented. Serverless microservices architecture and configuration driven backend-code. Minor changes like bug fixes can be solved by directly changing the configuration in database instead of

changing the code. And major issues like feature upgrades can be solved by just changing/adding the micro service for that specific feature instead of changing the entire code.

1.5 Problem Statement:

1.5.1:

EMRs, as useful they may be, are not as easy to use as a paper record. With tons of information and patient history overflowing from just a single page, it becomes extremely difficult for the Doctor to focus only on the required information while treating the patient.

1.5.2:

Deploying applications on servers can cost a fortune. Plus if the server is under-utilised, it is a waste of money. Redundant and Idle servers are a major financial leak for companies that deploy their own websites or applications.

This paper outlines the process of synthesising an enterprise level solution to extract enough information from the EMR on the screen to uniquely identify the patient. Along the way, this paper also deals with another problem in software development, i.e. breaking down the monolithic architecture of software solutions to speed up the development cycle of solutions, support scalability and reduce the cost of running websites and applications drastically. In this approach, the software is composed of small independent services that communicate over small well-orchestrated APIs. This microservice architecture makes it easy to try out new ideas and roll it back with the help of continuous integration and continuous delivery if something undesirable happens. After this microservice is up and running, this paper will also explain how to write a “*configuration driven*” backend code which makes it easier to do minor and sometimes major changes by manipulating configurations in the database instead of changing the deployed code and re-deploying it. The paper takes an interdisciplinary stance by working with image processing and software architecture to create one solution that solves four major problems.

Chapter 2:

Literature:

To make this configuration driven Backend microservice to extract Human names from the active window on the screen, with Serverless architecture implemented in it, we used the following services/tools:

1. AWS Lambda
2. Docker
3. EC2
4. Django Framework
5. MongoDB
6. Image Processing

2.1 AWS Lambda:

Lambda is a compute service that lets you run code without provisioning or managing servers. Lambda executes the code only when needed and scales automatically, from few requests per day to thousands every second. AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. You can use AWS Lambda to run your code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table; to run your code in response to HTTP requests using Amazon API Gateway. With these capabilities, you can use Lambda to easily build data processing triggers for AWS services like Amazon S3 and Amazon DynamoDB, process streaming data stored in Kinesis, or create your own back end that operates at AWS scale, performance, and security.

We can build serverless applications composed of functions that are triggered by events and automatically deploy them. AWS Lambda is an ideal compute platform for many application scenarios, provided that you can write your application code in languages supported by AWS Lambda, and run within the AWS Lambda standard runtime environment and resources provided by Lambda.

When your function is invoked, Lambda attempts to re-use the execution environment from a previous invocation if one is available. When Lambda executes your function, it provisions and manages the resources needed to run your Lambda function. When creating a Lambda Function, we specify the configuration information, such as the amount of memory and a maximum execution time that you want to allow for your lambda function. When a Lambda function is invoked, AWS Lambda launches an execution context based on the configuration settings you provide. The execution context is a temporary runtime environment that initializes any external dependencies of your Lambda function code, such as database connections or HTTP endpoints. This affords subsequent invocations better performance because there is no need to "cold-start" or initialize those external dependencies.

It takes time to set up an execution context and do the necessary "bootstrapping", which adds some latency each time the Lambda function is invoked. You typically see this latency when a Lambda function is invoked for the first time or after it has been updated because AWS Lambda tries to reuse the execution context for subsequent invocations of the Lambda function.

After a Lambda function is executed, AWS Lambda maintains the execution context for some time in anticipation of another Lambda function invocation. In effect, the service freezes the execution context after a Lambda function completes, and thaws the context for reuse, if AWS Lambda chooses to reuse the context when the Lambda function is invoked again. This execution context reuse approach has the following implications:

1. Objects declared outside of the function's handler method remain initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations.
2. Each execution context provides 512 MB of additional disk space in the /tmp directory. The directory content remains when the execution context is frozen, providing transient cache that can be used for multiple invocations.
3. Background processes or callbacks initiated by your Lambda function that did not complete when the function ended resume if AWS Lambda chooses to reuse the execution context. You should make sure any background processes or callbacks in your code are complete before the code exits.

2.2 Docker:

1. Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a hypervisor, but run directly within the host machine's kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines.

Docker provides tooling and a platform to manage the lifecycle of your containers:

1. Develop your application and its supporting components using containers.

2. The container becomes the unit for distributing and testing your application.
3. When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker Engine:

Docker Engine is a client-server application with these major components:

1. A server which is a type of long-running program called a daemon process (the `dockerd` command).
2. A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
3. A command line interface (CLI) client (the `docker` command).

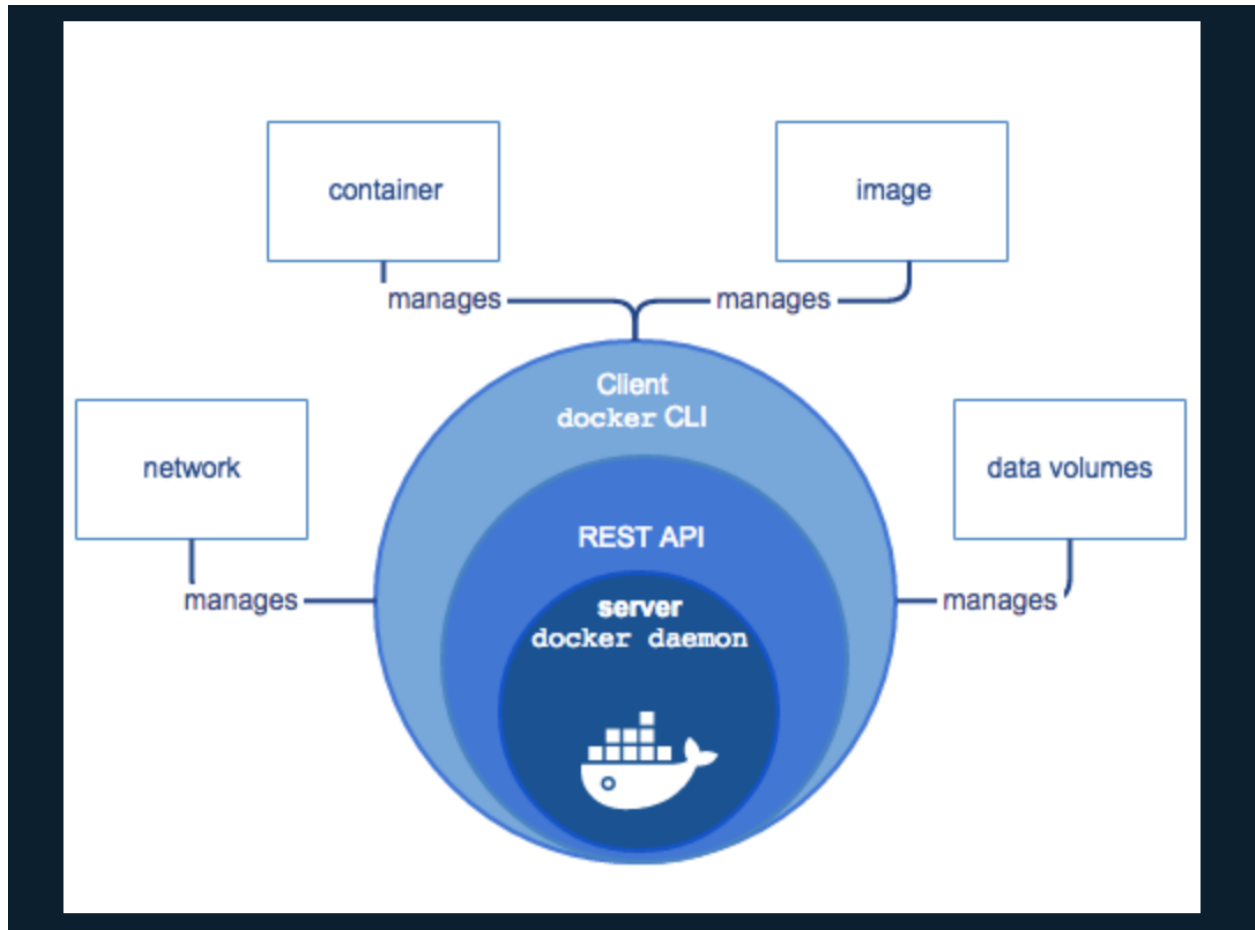


Fig.2: Docker Engine

The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI. The daemon creates and manages Docker objects, such as images, containers, networks, and volumes.

Fast, consistent delivery of your applications:

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Responsive deployment and scaling:

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

Docker's portability and lightweight nature also makes it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

Running more workloads on the same hardware:

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your compute capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

Docker architecture:

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

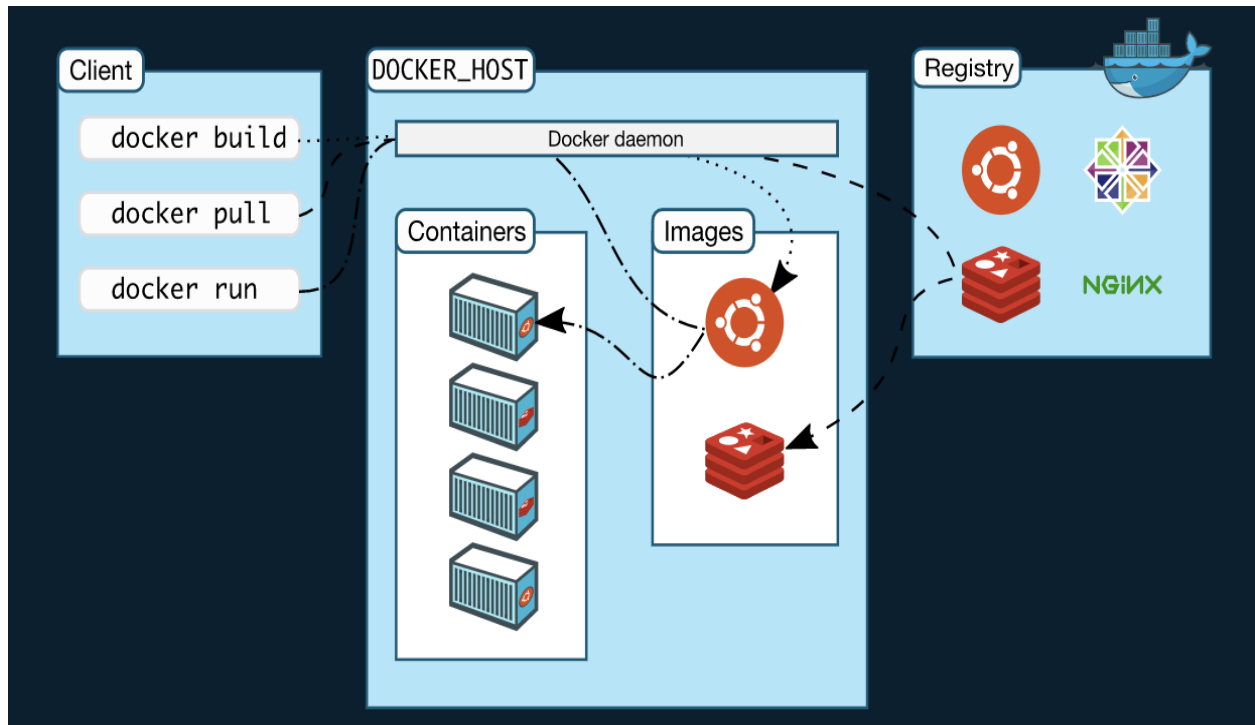


Fig.3: Docker Architecture

The Docker daemon:

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker Client:

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Registries:

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

IMAGES:

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

CONTAINERS:

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

2.3 Amazon EC2:

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) cloud.



Fig.4: EC2 can be accessed through SSH

Amazon EC2 provides the following features:

1. Virtual computing environments, known as instances
2. Preconfigured templates for your instances, known as Amazon Machine Images (AMIs), that package the bits you need for your server (including the operating system and additional software)
3. Various configurations of CPU, memory, storage, and networking capacity for your instances, known as instance types
4. Secure login information for your instances using key pairs (AWS stores the public key, and you store the private key in a secure place)
5. Storage volumes for temporary data that's deleted when you stop or terminate your instance, known as instance store volumes
6. Persistent storage volumes for your data using Amazon Elastic Block Store (Amazon EBS), known as Amazon EBS volumes
7. Multiple physical locations for your resources, such as instances and Amazon EBS volumes, known as Regions and Availability Zones

2.4 Django Framework:

Django is a web development framework that assists in building and maintaining quality web applications. Django helps eliminate repetitive tasks making the development process an easy and time saving experience. Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Django makes it easier to build better web apps quickly and with less code.

History of Django:

1. 2003 – Started by Adrian Holovaty and Simon Willison as an internal project at the Lawrence Journal-World newspaper.
2. 2005 – Released July 2005 and named it Django, after the jazz guitarist Django Reinhardt.
3. 2005 – Mature enough to handle several high-traffic sites.
4. Current – Django is now an open source project with contributors across the world.

Django – Design Philosophies:

Django comes with the following design philosophies –

1. Loosely Coupled – Django aims to make each element of its stack independent of the others.
2. Less Coding – Less code so in turn a quick development.
3. Don't Repeat Yourself (DRY) – Everything should be developed only in exactly one place instead of repeating it again and again.
4. Fast Development – Django's philosophy is to do all it can to facilitate hyper-fast development.
5. Clean Design – Django strictly maintains a clean design throughout its own code and makes it easy to follow best web-development practices.

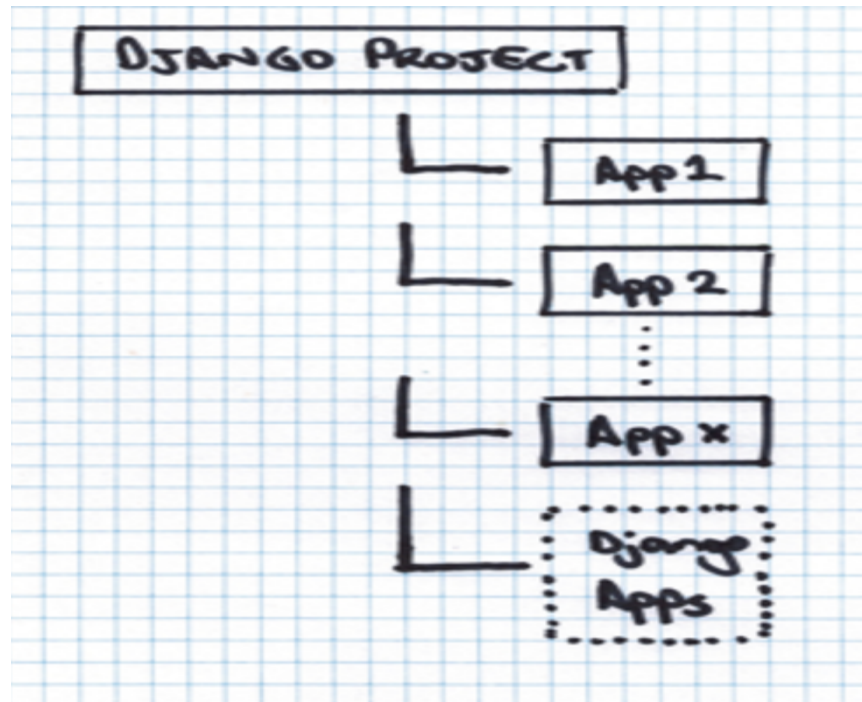


Fig.5: Structure of a Django Project

Advantages of Django

Here are few advantages of using Django which can be listed out here –

1. Object-Relational Mapping (ORM) Support – Django provides a bridge between the data model and the database engine, and supports a large set of database systems including MySQL, Oracle, Postgres, etc. Django also supports NoSQL database through Django-nonrel fork. For now, the only NoSQL databases supported are MongoDB and google app engine.
2. Multilingual Support – Django supports multilingual websites through its built-in internationalization system. So you can develop your website, which would support multiple languages.
3. Framework Support – Django has built-in support for Ajax, RSS, Caching and various other frameworks.
4. Administration GUI – Django provides a nice ready-to-use user interface for administrative activities.

5. Development Environment – Django comes with a lightweight web server to facilitate end-to-end application development and testing.

MVC Pattern:

When talking about applications that provides UI (web or desktop), we usually talk about MVC architecture. And as the name suggests, MVC pattern is based on three components: Model, View, and Controller.

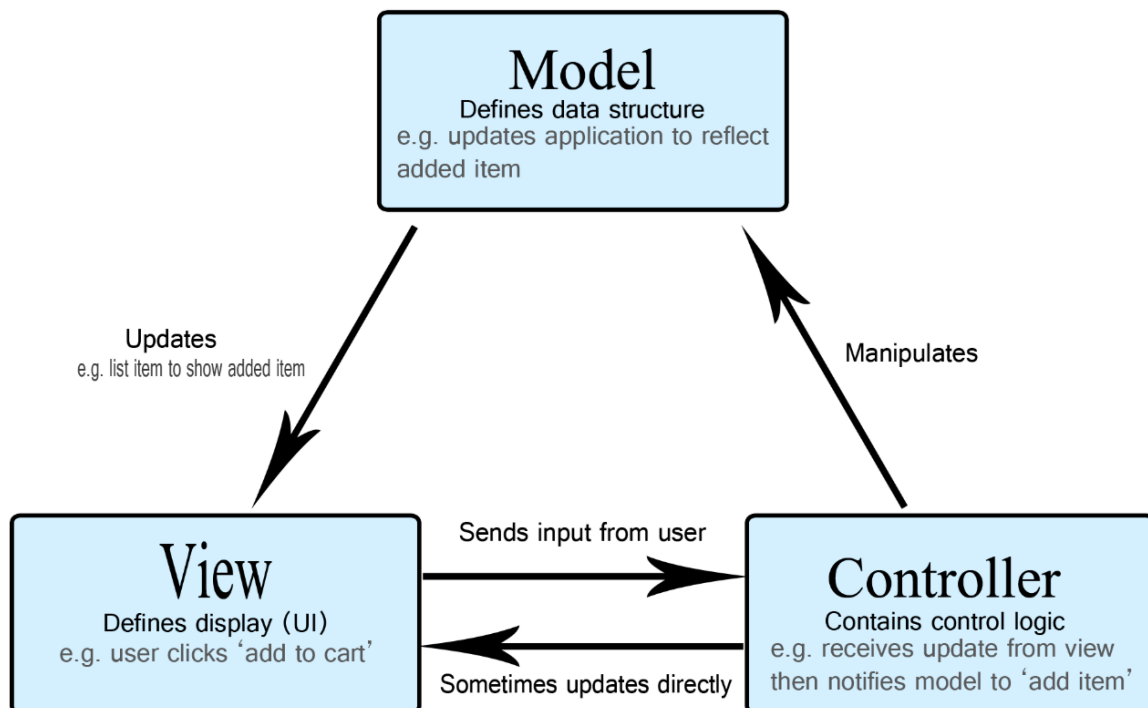


Fig.6: MVC Pattern

The Model:

The model defines what data the app should contain. If the state of this data changes, then the model will usually notify the view (so the display can change as needed) and sometimes the controller (if different logic is needed to control the updated view).

The View:

The view defines how the app's data should be displayed.

The Controller:

The controller contains logic that updates the model and/or view in response to input from the users of the app.

Django MVC - MVT Pattern:

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template. The template is an HTML file mixed with Django Template Language (DTL).

The following diagram illustrates how each of the components of the MVT pattern interacts with each other to serve a user request –

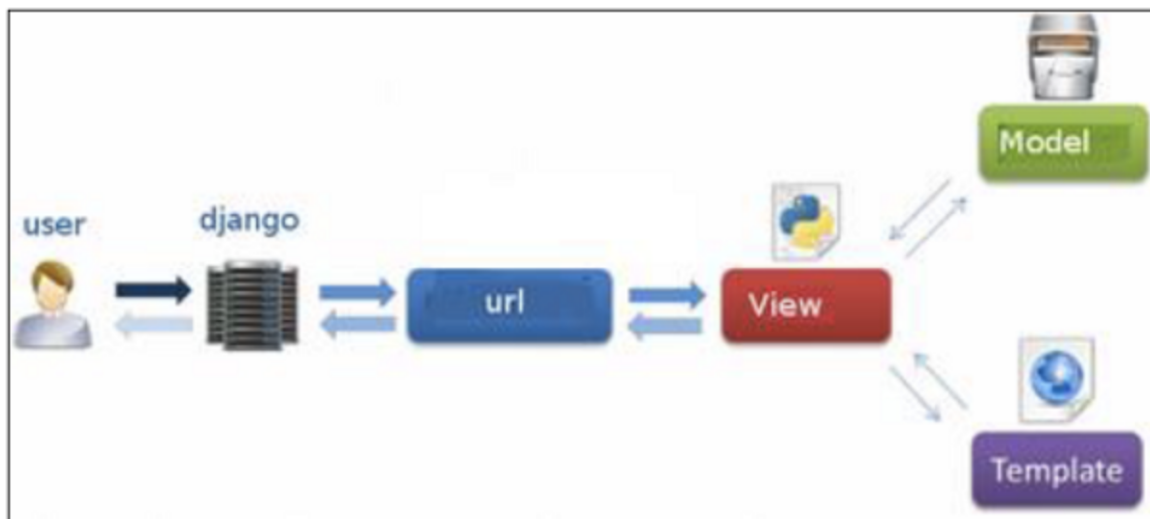


Fig.7: MVT Architecture in Django

As from the diagram below, we have some components and two regions i.e., server side and client side. Here you will notice that the View is on the server-side part while the template is on the client side.

Now, when we request for the website, the interface through which we use to make that request via our browser was the Template. Then that request transmits to the server for the management of view file.

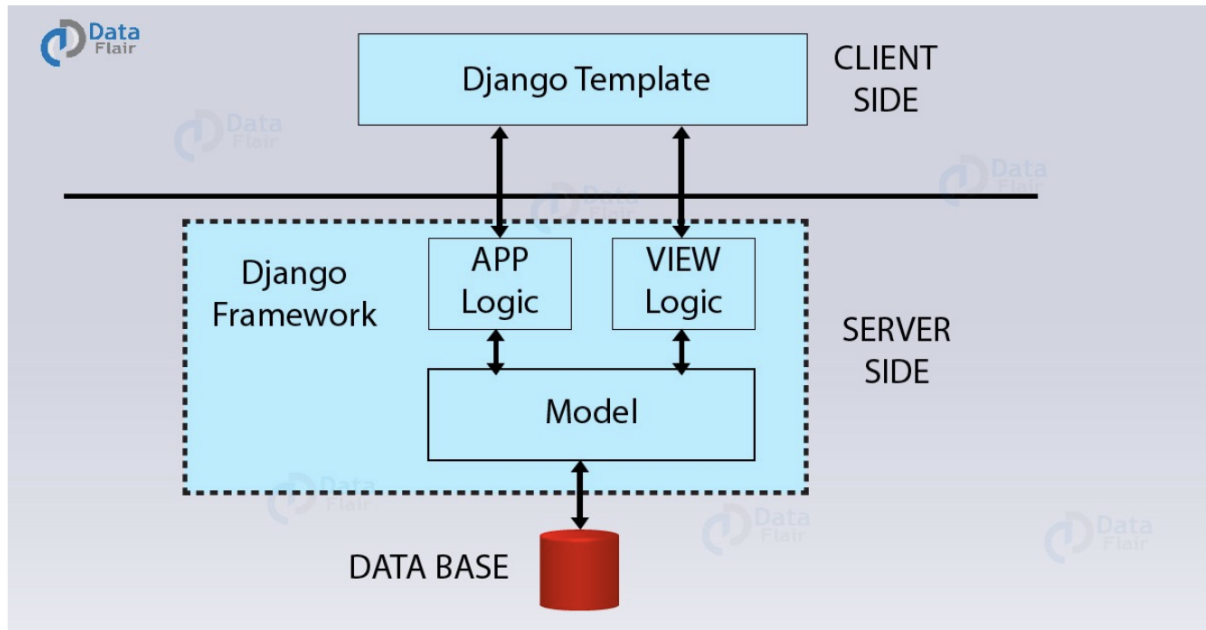


Fig.8: Architecture of Django

Django is literally a play between the requests and responses. So whenever our Template is updating it's the input (request) we sent from here which on the server was seen by the View. And, then it transports to the correct URL. It's one of the important components of Django MTV architecture. There, the URL mapping in Django is actually done in regular expressions. These expressions are much more understandable than IP addresses. It also helps with the SEO task which we have discussed in the Django Features Tutorial.

Now after the sending of a request to the correct URL, the app logic applies and the model initiates to correct response to the given request. Then that particular response is sent back to the View where it again examines the response and transmits it as an HTTP response or desired user format. Then, it again renders by the browser via Templates.

Web Server:

Django comes with a lightweight web server for developing and testing applications. This server is pre-configured to work with Django, and more importantly, it restarts whenever you modify the code.

2.5 MongoDB:

MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era. MongoDB is a document database, which means it stores data in JSON-like documents.

Features of MongoDB:

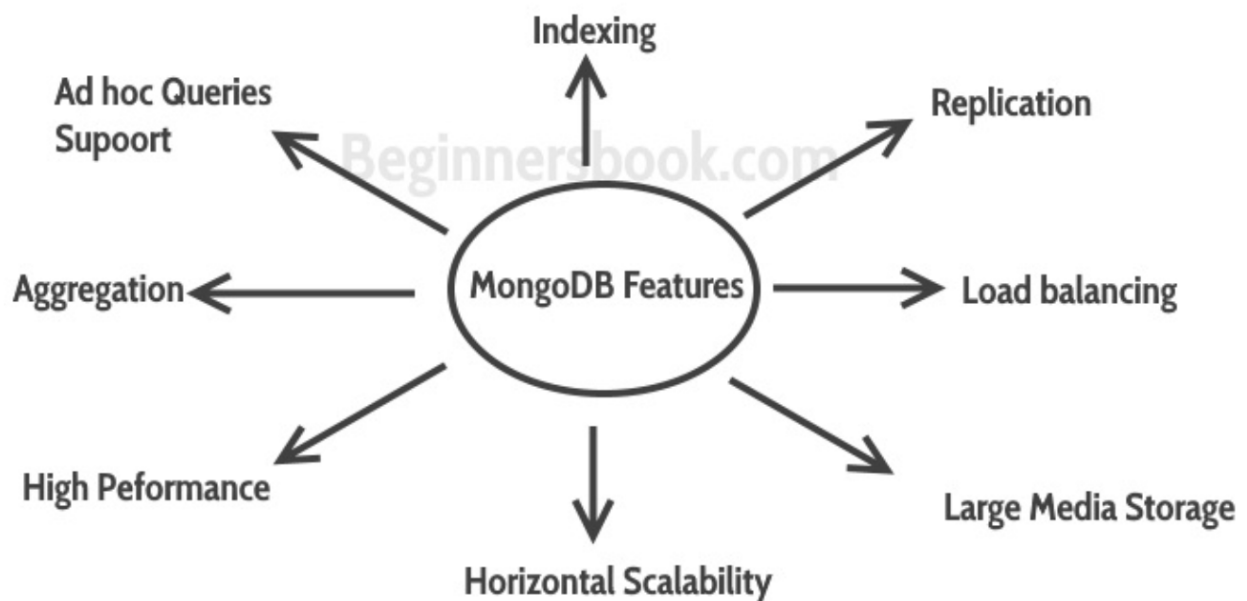


Fig.9: Features of MongoDB

1. MongoDB provides high performance. Most of the operations in the MongoDB are faster compared to relational databases.
2. MongoDB provides auto replication feature that allows you to quickly recover data in case of a failure.

3. Horizontal scaling is possible in MongoDB because of sharing. Sharding is partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved.
4. Load balancing: Horizontal scaling allows MongoDB to balance the load.
5. High Availability: Auto Replication improves the availability of MongoDB database.
6. Indexing: Index is a single field within the document. Indexes are used to quickly locate data without having to search every document in a MongoDB database. This improves the performance of operations performed on the MongoDB database.

Horizontal scaling vs vertical scaling:

Vertical scaling means adding more resources to the existing machine while horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement, on the other hand, horizontal scaling is easy to implement. Horizontal scaling database examples: MongoDB, Cassandra etc.

Mapping relational Database to MongoDB:

RDBMS stands for Relational Database Management Systems. A relational database refers to a database that stores data in a structured format, using rows and columns. This makes it easy to locate and access specific values within the database. It is "relational" because the values within each table are related to each other. Tables may also be related to other tables. The relational structure makes it possible to run queries across multiple tables at once.

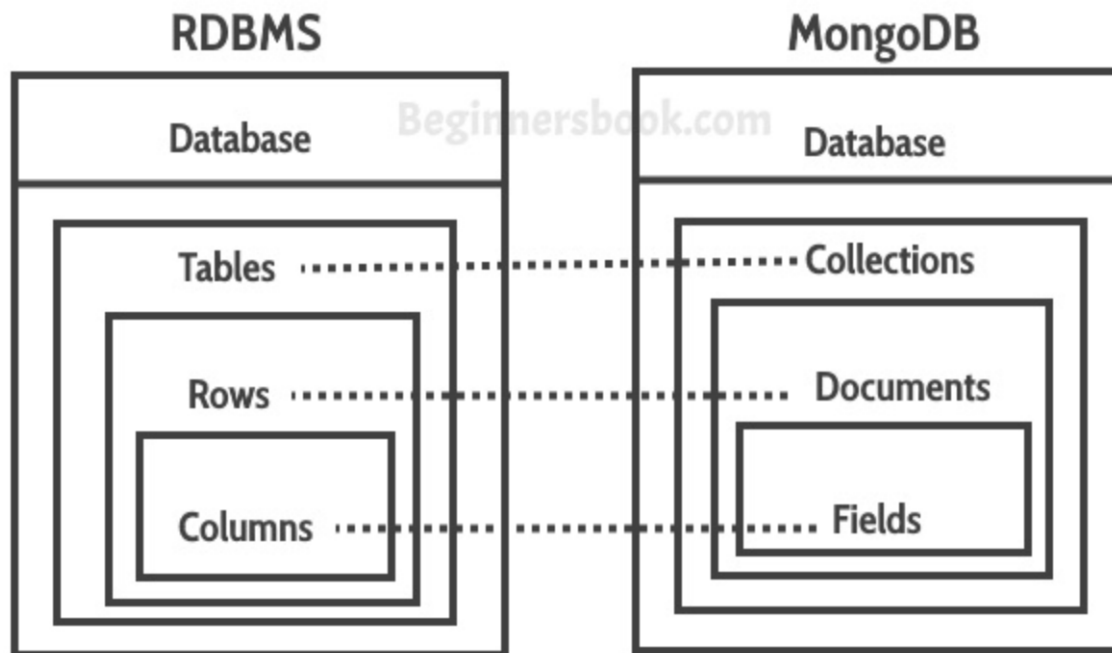


Fig.10: Comparison Between RDBMS and MongoDB

Collections in MongoDB is equivalent to the tables in RDBMS.

Documents in MongoDB is equivalent to the rows in RDBMS.

Fields in MongoDB is equivalent to the columns in RDBMS.

2.6 Image Processing:

Digital image processing is to process images by computer. Digital image processing can be defined as subjecting a numerical representation of an object to a series of operations in order to obtain a desired result. Digital image processing consists of the conversion of a physical image into a corresponding digital image and the extraction of significant information from the digital image by applying various algorithms. Digital image processing mainly includes image collection, image processing, and image analysis

At its most basic level, a digital image processing system is comprised of three components, ie, a computer system on which to process images, an image digitizer, and an image display device. Fig. 11 shows a complete system for image processing.

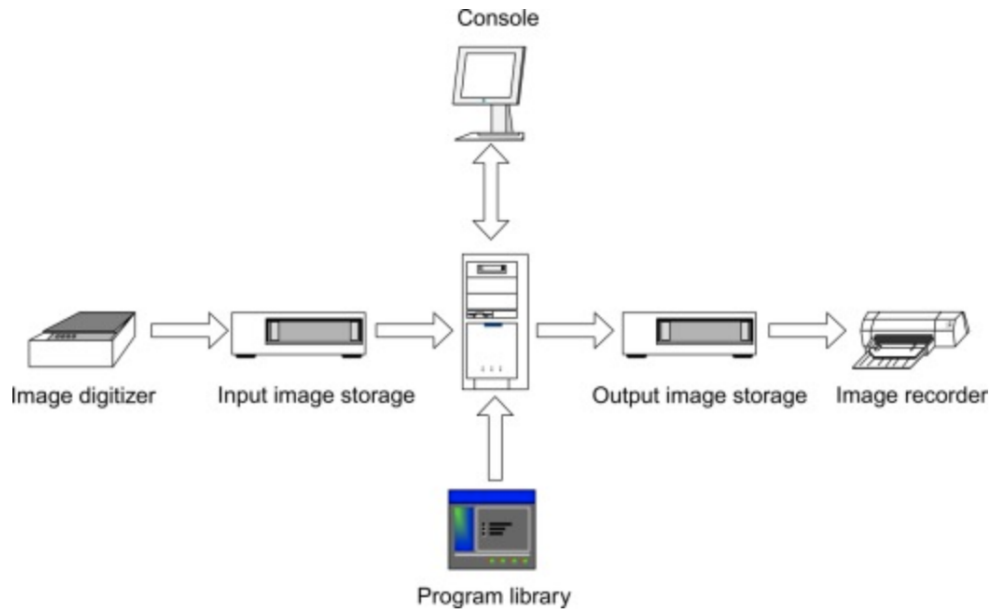


Fig.11: Complete system for Image Processing

Physical images are divided into small areas called pixels. The division plan used often is the rectangular sampling grid method shown in Fig. 12, in which an image is segmented into many horizontal lines composed of adjacent pixels, and the value of each pixel position reflects the brightness of corresponding point on the physical image.

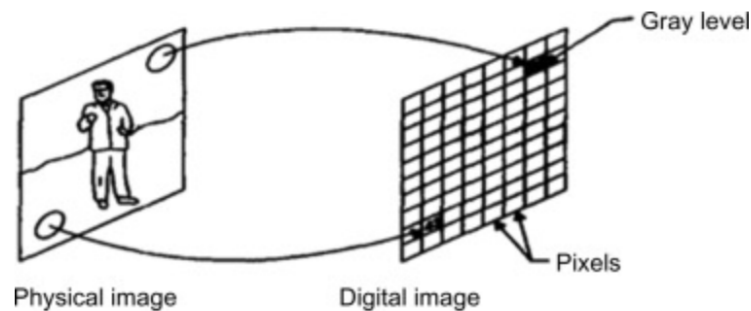


Fig.12: Division Plan

Physical images cannot be directly analyzed by a computer because the computer can only process digits rather than images, so an image must be converted into a digital form before processed by a computer. The conversion process is called digitization, and a common form is shown in Fig. 13

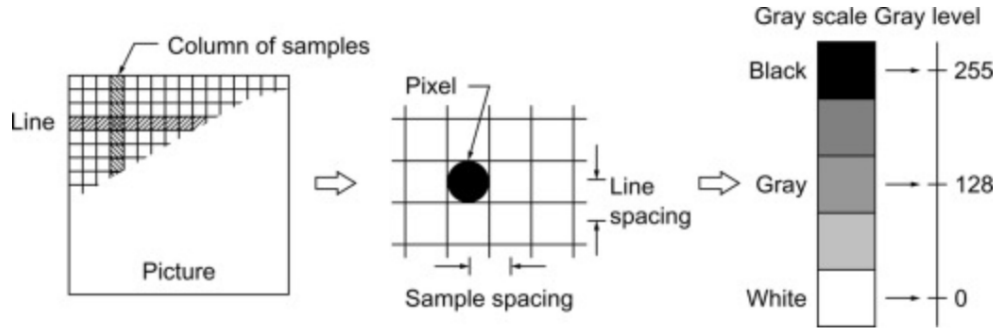


Fig.13: Digitization

At each pixel position, the brightness is sampled and quantized to obtain an integer value indicating the brightness of the corresponding position in the image. After the conversion of all pixels of an image is completed, the image can be represented by a matrix of integers. Each pixel has two attributions: position and gray level. The position is determined by the two coordinates of sampling point in the scanning line, namely row and column. The integer indicating the brightness of the pixel position is called gray level.

Images displayed by digital matrix are called digital images, and all digital image processing is based on the digital matrix. The digital matrix is the object processed by a computer.

$$F = \begin{bmatrix} f(0, 0) & f(0, 1) & \dots & f(0, N - 1) \\ f(1, 0) & f(1, 1) & \dots & f(1, N - 1) \\ \dots & \dots & \dots & \dots \\ f(M - 1, 0) & f(M - 1, 1) & \dots & f(M - 1, N - 1) \end{bmatrix}$$

where

F = image

$f(i, j)$ = the gray level of pixel (i, j) .

Fig.14: Digital Matrix

On the basis of image processing, it is necessary to separate objects from images by pattern recognition technology, then to identify and classify these objects through technologies provided

by statistical decision theory. Under the conditions that an image includes several objects, the pattern recognition consists of three phases, as shown in Fig. 15

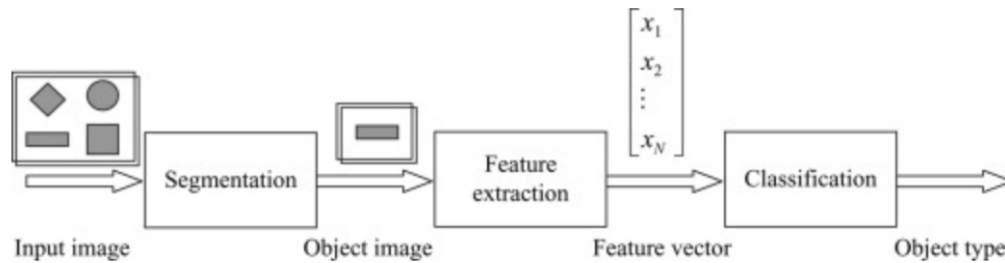


Fig.15: Three Phases of pattern recognition.

The first phase includes the image segmentation and object separation. In this phase, different objects are detected and separate from other background. The second phase is the feature extraction. In this phase, objects are measured. The measuring feature is to quantitatively estimate some important features of objects, and a group of the features are combined to make up a feature vector during feature extraction. The third phase is classification. In this phase, the output is just a decision to determine which category every object belongs to. Therefore, for pattern recognition, what input are images and what output are object types and structural analysis of images. The structural analysis is a description of images in order to correctly understand and judge for the important information of images.

Before we can develop predictive models for image data, we must learn how to load and manipulate images and photographs. The most popular and de facto library in Python for loading images and working with images is Pillow. Pillow is an updated version of Python Image Library or PIL, and supports a range of simple and sophisticated image manipulation functionality. It is also the basis for simple image support in other Python libraries such as SciPy and Matplotlib.

Images are typically in PNG and JPEG format and can be loaded directly using the `open()` function on Image class. This returns an Image Object that contains the pixel data for the image as well as details about the image. The Image class is the main workhorse for the Pillow library and provides a ton of properties about the image as well as functions that allow you to manipulate pixels and format of the image. The `format` property on the image will report the image format and the `'mode'` will report the pixel channel format (RGB or CMYK), and the `'size'` will report the dimensions of the image in pixels. The `show()` function will display the image using operating systems default application.

Often in machine learning we want to work with images as NumPy arrays of pixel data. With Pillow installed, you can also use the Matplotlib library to load the image and display it within a frame. This can be achieved using the `imread()` function that loads the image an array of pixels directly and `imshow()` function that will display an array of pixels as an image.

Understanding the pixel grid

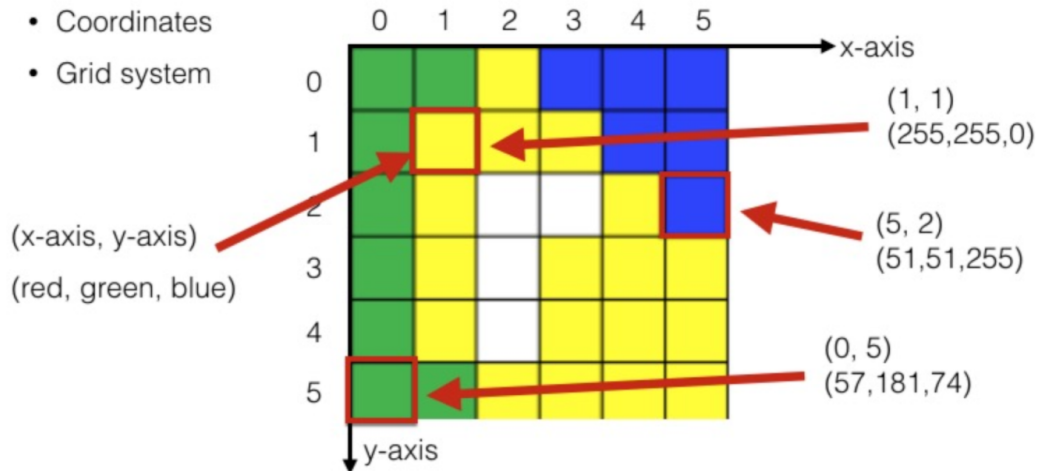


Fig.16: Pixel Grid

Chapter 3:

Methodology/Experiment:

Approach:

1. To tackle with redundant idle servers, we first deployed a Lambda function which will implement “*Serverless*” architecture in our product.
2. Once the Lambda function is up and running, we need a way to detect if an EMR is active on the screen of the user. If active, we will take a screenshot of the active screen. We will crop the screenshot in order to optimise the parsing algorithm and reduce latency, as Optical Character Recognition takes quite the amount of time as compared to other steps. We will send this cropped image to the Lambda function which will use the OCR’s API to extract text out of this image. Once we are able to determine the text, we will extract the name of the patient from this text.
3. We will write an API using Django in such a way that it is driven by configuration in our database (MongoDB). In this way, we will be able to control and manipulate the behaviour of the code easily as compared to changing the code of the API. Then we will use our configuration driven backend to search for the patient in our database. If found, we will show vital information such as Care Gaps, Current and Past Medications on the screen.
4. After the above steps are done, we will build a continuous Integration/Deployment pipeline that will automate building and deploying both our Lambda function and backend service whenever we change the code.

3.1: Deploying a Lambda function:

Difficulties faced while implementing serverless architecture through Lambda:

1. Size limit on dependencies package.
2. Environment compatibility Issue.

3.1.1 Size Limit of Python Packages:

On getting invoked, the Lambda function will use modules and packages as dependencies to run the code. We need to provide these dependencies as a prerequisite to the Lambda function. There are multiple ways to do this:

1. If the size of the package is less than 10MB it can be directly uploaded at the time of building and configuring the Lambda function.
2. If the size exceeds 10MB but is less than or equal to 50MB, it can still be uploaded directly but the code cannot be edited at the time of configuring. If any changes are to be made, it has to be done by changing the file that contains the source code and uploading it once again.
3. If the size exceeds 50MB, it can be uploaded in two ways:
 - a. The deployment package can be uploaded to Amazon S3 Bucket and the path to the S3 Bucket can be given to the Lambda function. On getting invoked, the lambda will get the package from the given path, start execution of the handler defined in the lambda function as use dependencies from the same package to run the code.
 - b. Lambda provides the service of adding layers. You can configure lambda to pull in additional code and content in the form of layers. A layer is a ZIP archive that contains libraries or dependencies. With layers, you can use the libraries in your function without needing to include them in your deployment package. Layers let you keep your deployment package small, which makes deployment easier. But layers have a limitation. A function can use up to 5 layers at a time. The total unzipped size of the function and all layers can't exceed the unzipped deployment package size limit of 250MB.

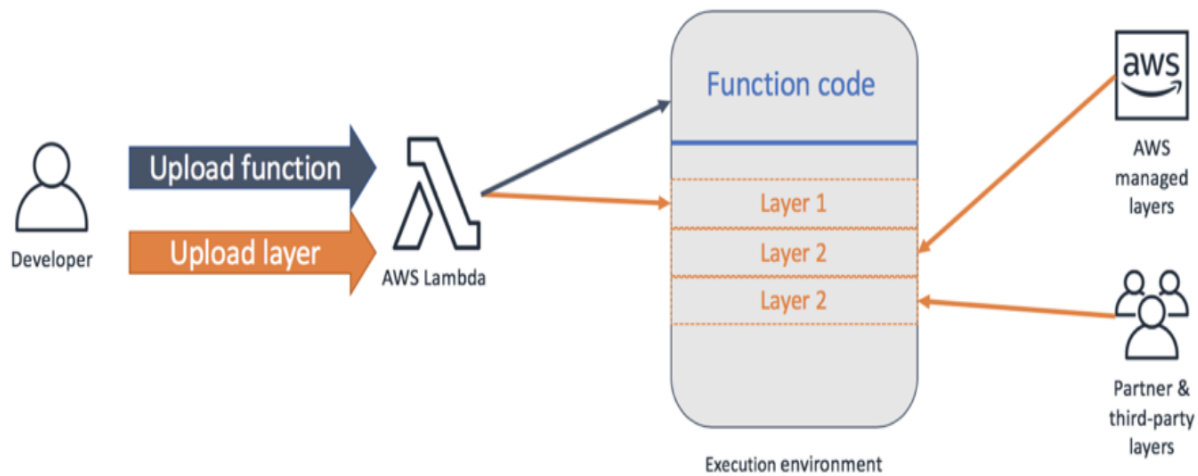


Fig.17: Explaining Layers in Lambda

Since we'll be using dependencies and libraries that have a total size of more than 50 MBs, we will use an Amazon S3 bucket to store our deployment package.

3.1.2 Execution Environment Compatibility:

The next problem faced with deploying the code is environment compatibility. We'll be using libraries that have third-party dependencies. C and C++ extensions need to be compiled before using these libraries. The compilation part is taken care of at the time of importing the library/module. When importing any package, if the package has dependencies on these extensions, these extensions will get compiled. And only then these packages can be used. On the user (developer) level, when we install any package which is to be used in the code, these third-party dependencies are stored and the path to these dependencies is set in the main package. When the main package gets imported, it first gets to these third-party packages through the given path and compiles them before getting imported in the code. This "path" to these extensions varies with the development environment. This is where the problems occur. If the environment in which we develop the deployment package before uploading the ZIP file to AWS isn't the same as that on AWS Lambda, the third-party dependencies will be on a different path and thus will not get compiled. As a result, the main package, which uses these extensions and which is to be used in the code, will not get imported. This means that the lambda function cannot execute the code.

We can replicate the development environment that AWS Lambda uses to make the deployment package. This will keep the third-party dependencies on the same path as that in the main packages that compile them. There are a number of ways of doing this:

1. We can use an AWS EC2 AMI to install the libraries and make the deployment packages. Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) cloud. Using Amazon EC2 eliminates your need to invest in hardware up front, so you can develop and deploy applications faster. You can use Amazon EC2 to launch as many or as few virtual servers as you need, configure security and networking, and manage storage. Amazon EC2 enables you to scale up or down to handle changes in requirements or spikes in popularity, reducing your need to forecast traffic.

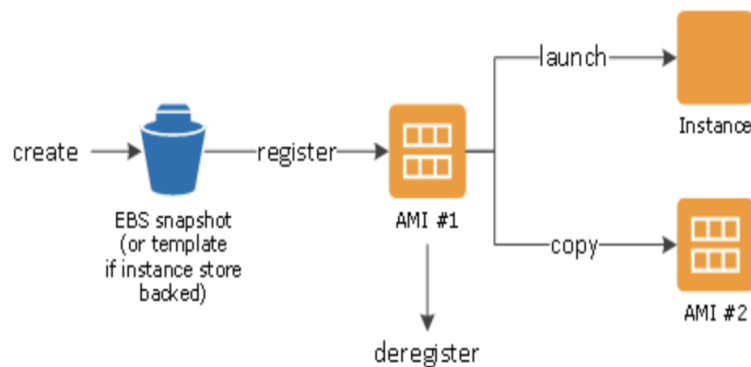


Fig. 18: How AMI is launched

AMI stands for Amazon Machine Image. It is a template that contains a software Configuration (for example, an operating system, an application server, and applications). From an AMI, you launch an *instance*, which is a copy of the AMI running as a virtual server in the cloud. You can launch multiple instances of an AMI, as shown in the following figure.

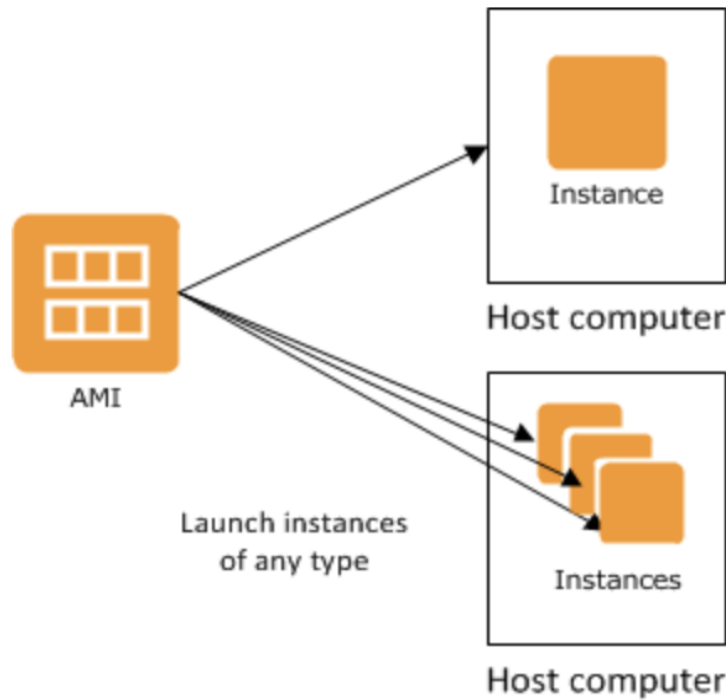


Fig.19: How Instances are Run

Instances: An instance is a virtual server in the cloud. Its configuration at launch is a copy of the AMI that you specified when you launched the instance. You can launch different types of instances from a single AMI. An *instance type* essentially determines the hardware of the host computer used for your instance. Each instance type offers different compute and memory capabilities. Select an instance type based on the amount of memory and computing power that you need for the application or software that you plan to run on the instance.

After you launch an instance, it looks like a traditional host, and you can interact with it as you would any computer. You have complete control of your instances; you can use **sudo** to run commands that require root privileges.

AWS publishes many AMIs that contain common software configurations for public use. In addition, members of the AWS developer community have published their own custom AMIs. You can also create your own custom AMI or AMIs; doing so enables you to quickly and easily start new instances that have everything you need.

AWS Lambda runs on top of the Amazon Linux operating system distribution and maintains updates to both the core OS and managed language runtimes. Therefore we need

to launch and EC2 AMI instance of similar execution environment. After this, we can install all the libraries, make a ZIP archive file and upload this deployment package to S3 Bucket. By doing this, the deployment package will be made in the same execution environment in which it will be executed. Therefore we won't face any compatibility issues.

2. We can use Docker to run a loosely isolated container and make our deployment package inside it. Thus we can build or use a pre-existing docker image that has an execution environment which is similar to that in AWS Lambda. We can use this image to run a container and install our dependencies within this container to make the deployment package.
3. We can use a framework called "*Serverless*", which is an open source command line tool that makes it easy to develop, deploy and test serverless apps across different platforms. Serverless uses docker images in the same way as mentioned in preceding methods. It provides an easier way to deploy a Lambda function as the user doesn't have to worry about building and running a docker image.



Fig.20: Deploying Lambda with S3 Bucket and API Gateway, using Serverless

Pros and Cons of Serverless architecture:

Pros of serverless computing:

1. Cheaper than the traditional Cloud - It allows you to pay a fraction of the price per request. Startups can build a feature nearly for free and ease into the market without dealing with huge bills for minimum traffic.

2. Scalable - It scales automatically according to the traffic.

3. Lower human resources costs - Just as you don't have to pay thousands of dollars on hardware, you can stop paying engineers for maintaining it.

Cons:

1. Vendor Lock-in - You have to play by the rules of the vendor. Right now only Node.js and Python developers have the freedom to choose between existing serverless options.

2. Learning Curve - Despite the comprehensive documentation, you may soon find that the learning curve for faas(function as a service) is pretty steep.

3. Unsuitable for long term tasks - Lambda gives you 5 minutes to execute a task and if it takes longer, you'll have to call another function.

3.2 Text Extraction:

Difficulties:

1. Detect that an EMR is active.
2. Capture screenshot and crop it according to the window size.
3. Extract Text using OCR.

3.2.1 EMR Detection:

To detect if an EMR is active, we will use a python module *win32gui* which is a wrapper on python's built-in *ctypes* module, which is also a wrapper on C and C++ extensions.

```
def get_active_window():
    try:
        window_handle = win32gui.GetForegroundWindow()
        active_window_name = win32gui.GetWindowText(window_handle)
        return active_window_name, window_handle
    except:
        pass
```

Fig.21: Function to determine the name of the active window on screen.

3.2.2 Capturing and cropping screenshot:

After we have the name of the active window on the screen, if this window is an EMR, we have to capture the screenshot and crop it.

We write a program to take a screenshot of the active window and crop it.

First we determine if a specific window is active. Once we find that window to be on foreground, we get the handle to that window by using a python module named win32gui. “Win32gui” provides an interface to the native win32 GUI API. We pass this handle to another function that grabs the screenshot of this window.

```
def grab_screenshot(window_handle):
    try:
        monitors = dm.getDisplaysAsImages()
        if len(monitors) > 1:
            return get_multiple_monitors(window_handle, monitors)
        else:
            return get_single_monitor(window_handle)
    except Exception as e:
        pass
```

Fig.22: Function to determine number of monitors and grab screenshot

Since the system can have multiple monitors and our window can be active on any one of them, we need to determine which monitor has our window on Foreground. For that we first find out the number of monitors on the system. If only one monitor is detected, we simply call a function that will take a screenshot of that monitor. If there are more than two monitors, we use win32gui to find the position of the application window with the top left corner of our primary window as origin. Depending on the coordinates of the application window (left, top, bottom, right) we determine where the secondary monitor is located with respect to the primary monitor. Once that is done, we now normalise the coordinates of the application window with respect to the monitor on which it is displayed on.


```

def get_multiple_monitors(window_handle, monitors):
    try:
        normalized_box = []
        box = win32gui.GetWindowRect(window_handle)
        primary_monitor = monitors[0]
        secondary_monitor = monitors[-1]

        if box[LEFT] > (primary_monitor.width-15):
            # Secondary monitor is on the Right
            normalized_box = [
                box[LEFT] - primary_monitor.width,
                box[TOP],
                box[RIGHT] - primary_monitor.width,
                box[BOTTOM]
            ]

        elif box[RIGHT] < 15:
            # Secondary monitor is on the Left
            normalized_box = [
                box[LEFT] + secondary_monitor.width,
                box[TOP],
                box[RIGHT] + secondary_monitor.width,
                box[BOTTOM]
            ]

        elif box[BOTTOM] > primary_monitor.height:
            # Secondary monitor is on the Bottom
            normalized_box = [
                box[LEFT],
                box[TOP] - primary_monitor.height,
                box[RIGHT],
                box[BOTTOM] - primary_monitor.height
            ]

```

Fig.23: Function to normalise Coordinates of the application window

Once we get the normalised coordinates, we use them to crop the screenshot of the secondary monitor and save the cropped image at a given path.

```
screenshot = secondary_monitor.crop(normalized_box)
screenshot.save(SCREENSHOT_PATH)
```

Fig.24: Code to grab and save screenshot of the application window

3.2.3 Text Extraction:

After we have a screenshot of the application window, we send a request to the Lambda function, that we deployed in the above steps, which will use OCR's API to extract out of that image and use custom algorithms to determine the first name and last name of a person from the text, if any.

```
def extract_image_text(bytes_image):
    '''Extract text from image using Google Vision API.

    Args:
        bytes_image: BufferedReader object

    Returns:
        stringified text
    '''
    client = ImageAnnotatorClient()

    image = types.Image(content=bytes_image)
    image_annotation_response = client.document_text_detection(image=image)
    annotations = image_annotation_response.text_annotations
    image_text = annotations[0].description.encode('ascii', 'ignore').decode()
    text = ' '.join(image_text.splitlines())
    return text
```

Fig.25: Extracting text from Image

Since we will be using this service to extract human names from screenshots of EMRs, we can write EMR specific algorithms to use in the experimental stage, and if it gets us any promising results we can use this service and optimise it to support scalability.

To write custom algorithms, we use Regular Expressions in python. A regular expression is a special sequence of characters that helps you match or find other strings or sets of

strings, using a specialized syntax held in a pattern. It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents. While using the regular expression the first thing to recognize is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters also referred as string

Use Regular Expressions in Python and some date parsing python modules to extract date of birth from the text. Most EMRs have the Patient's name in title case. And the rest of the text in normal case. We can write a regular expression which will look for Title cased words in the text and extract them and return them in the form of a list.

```
>>>
>>> import re
>>>
>>> pattern = '([A-Z][a-zA-Z]+),\s([A-Z][a-zA-Z]*) (\s)?([A-Z][a-zA-Z]+|[A-Z](.))??'
>>>
>>> regex = re.compile(pattern)
>>>
>>> matches = regex.findall(text)
```

Fig.26: Using regex to detect patterns in the text

Once we have this list, depending on the name format of the patient on a specific EMR, we can pick elements from the list as first name and last name. Since we have the freedom to write EMR specific custom logic, we can recognise patterns in the text of the EMRs and write algorithms which use these patterns to trim the text in order for regex to have better efficiency.

3.3 Writing a Configuration driven backend:

In this step we used Django to write an API which will search for patient name and DOB in the database. If found it will return the entire object corresponding to that patient in the mapping. The code behind this API will use configurations fetched from MongoDB to run functions and blocks of code, depending on the values of flag variables in the configuration.

```

"CHILDREN_MERCY": {
  "patientInWindowTitle": {
    "active": true,
    "pattern": "(( [A-Za-z]+)(-)?([A-Za-z]+))",
    "filterConfigs": {
      "necessaryKeywords": [],
      "blacklistedKeywords" : [
        "SEARCH",
        "REPORTS",
        "DASHBOARD",
        "RECENT",
        "CASE",
        "AUTHORIZATION"
      ],
      "thresholdFractions": {
        "necessary": 1,
        "blacklisted": 0.0
      }
    }
  },
  "detectPatientChart": {
    "active": true,
    "pattern": "[a-zA-Z]+",
    "filterConfigs" : {
      "necessaryKeywords" : [
        "ELIGIBILITY",
        "DOB",
        "DCN",
        "DEN"
      ],
      "blacklistedKeywords" : [],
      "thresholdFractions": {
        "necessary": 0.75,
        "blacklisted": 1
      }
    },
    "detectPatterns": {
      "active": false,
      "patterns": []
    }
  },
}

```

Fig.27: Configuration in MongoDB which determines the behaviour of the code

3.4 Building a CI/CD pipeline using Jenkins:

After implementing serverless architecture to make a microservice, we have to build a continuous integration(CI) and continuous deployment(CD) job. The adoption of CI/CD has changed how developers and testers ship softwares. Development teams have adapted to the shortened delivery cycles by embracing automation across their software delivery pipeline. Most teams have automated processes to check in code and deploy to new environments.

Continuous integration focuses on blending the work products of individual developers together into a repository. Often, this is done several times each day, and the primary purpose is to enable early detection of integration bugs, which should eventually result in tighter cohesion and more development collaboration. The aim of continuous delivery is to minimize the friction points that are inherent in the deployment or release processes. Typically, the implementation involves automating each of the steps for build deployments such that a safe code release can be done—ideally—at any moment in time. Continuous deployment is a higher degree of automation, in which a build/deployment occurs automatically whenever a major change is made to the code.

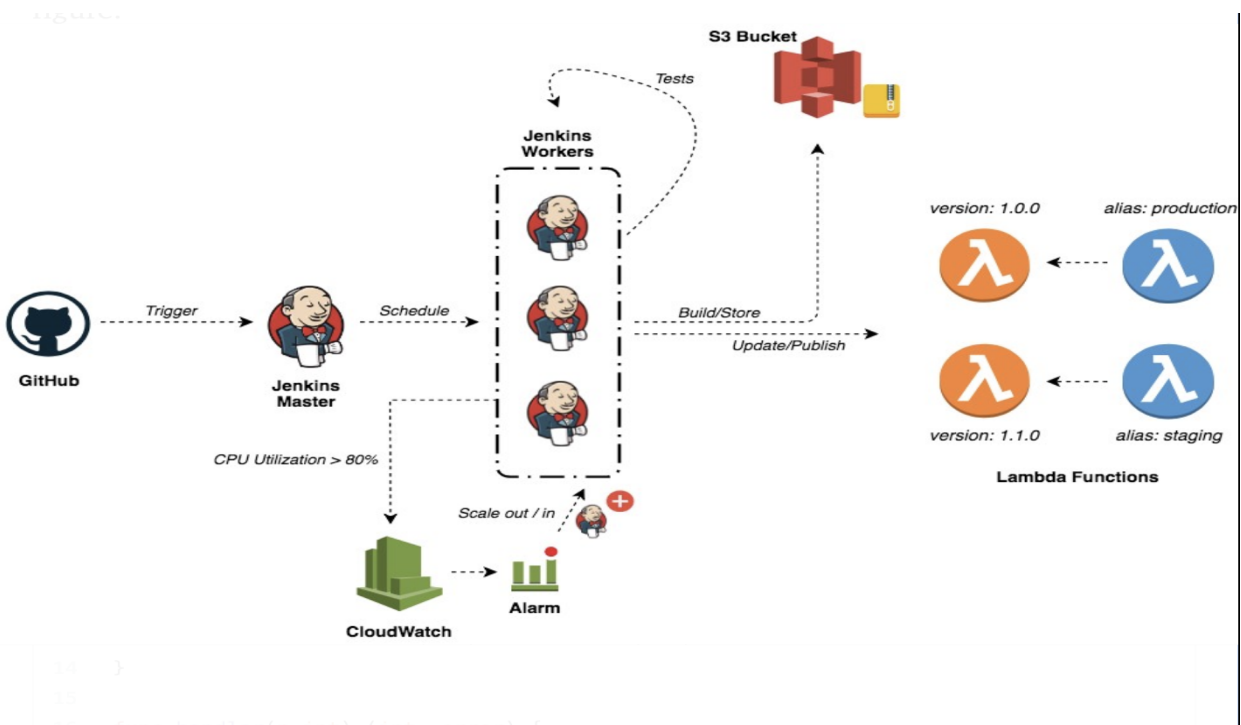


Fig.28: CI/CD integration with Jenkins explained

Continuous Integration (CI)

With continuous integration, developers frequently integrate their code into a main branch of a common repository. Rather than building features in isolation and submitting each of them at the end of the cycle, a developer will strive to contribute software work products to the repository several times on any given day.

The big idea here is to reduce integration costs by having developers do it sooner and more frequently. In practice, a developer will often discover boundary conflicts between new and existing code at the time of integration. If it's done early and often, the expectation is that such conflict resolutions will be easier and less costly to perform.

Continuous delivery is actually an extension of CI, in which the software delivery process is automated further to enable easy and confident deployments into production—at any time. A mature continuous delivery process exhibits a codebase that is always deployable—on the spot. With CD, software release becomes a routine event without emotion or urgency.

Continuous Deployment (CD)

Continuous deployment extends continuous delivery so that the software build will automatically deploy if it passes all tests. In such a process, there is no need for a person to decide when and what goes into production. The last step in a CI/CD system will automatically deploy whatever build components/packages successfully exit the delivery pipeline. Such automatic deployments can be configured to quickly distribute components, features, and fixes to customers, and provide clarity on precisely what is presently in production.

Organizations that employ continuous deployment will likely benefit from very quick user feedback on new deployments. Features are quickly delivered to users, and any defects that become evident can be handled promptly. With the movement to DevOps, there's also been a rise of new automation tools to help with the CI/CD pipeline. These tools typically integrate with various developer tools including code repository systems like Github and bug tracking systems like Jira.

The most popular automation tool is Jenkins (formerly Hudson), which is an open source project supported by hundreds of contributors as well as a commercial company, Cloudbees. It is a Continuous Integration server capable of orchestrating a chain of actions that help to achieve the Continuous Integration process (and not only) in an automated fashion. It is a server-based application and requires a web server like Apache Tomcat. The reason Jenkins became so popular is that of its monitoring of repeated tasks which arise during the development of a

project. For example, if your team is developing a project, Jenkins will continuously test your project builds and show you the errors in early stages of your development.

We used Jenkins to write a CI/CD pipeline and integrated it with gitlab. Whenever a new commit is pushed to our repository in gitlab, the pipeline is run and the code is deployed on the lambda function with the updated deployment package. To go this we our pipeline would poll the gitlab repository. When we push a new commit, the webhook will get Jenkins to run the pipeline. The pipeline is written in a “*JenkinsFile*” and is kept in the repository. Jenkins will look for the JenkinsFile in the repository, and if found, it will run the code/pipeline written in it.

We can define different stages in the pipeline. In our pipeline we defined the first stage in order to run unit tests on the code. If any of the test cases failed, the pipeline will stop and latest code will not be deployed.

If the first stage clears, in the second stage we will be installing the required libraries and make the deployed packages. There are two ways of deploying the code. We can run a docker Image in the second stage, install dependencies, compress them to make a zip archive and upload/update the package in the S3 Bucket using AWS’s command line tools. Or we can use Serverless(framework) in the second stage to deploy the code. It will take care of steps such as deploying code and managing docker container by itself.

Stage 1:

```
stage("UnitTest") {
    steps {
        sh '''
        pip install --user -r requirements.txt
        python test.py

        '''
    }
}
```

Stage 2:

```
stage("Serverless deploy") {
    steps {
        sh '''
        #!/bin/bash
        npm install serverless
```

```

serverless config credentials --provider aws --key "USER KEY" --secret "SECRET
KEY" -o
serverless deploy -v

'''
}
}

```

In the first stage, we install the required libraries using PIP(python package management system). Then we run custom unit test cases, written in our python program - "test.py". We used the built-in unit testing framework in python, called UNIT TEST. If any of the test cases fail, the pipeline will stop executing and code will not be deployed on lambda.

In the second stage, we install Serverless(framework) using npm. NPM is the package manager for Node.js. We put the credentials for the AWS account through which the Lambda function is linked. Then we use the command to deploy the code on Lambda. Serverless contains an XML file which contains the configuration for the deployment. It uses that configuration file to deploy the code on the lambda and integrate it with API Gateway and returns an endpoint. In case the lambda function already exists with a pre-existing with endpoint, it will update the package and will not change the endpoint.

By Following the aforementioned steps, we can create a lambda function with continuous integration and delivery.

Pros of Jenkins:

1. Jenkins is open source and free - Jenkins is free to download and the source code is also available. This has spawned a growing community of developers who are actively helping each other and contributing to the Jenkins project. This ensures that you get better and more stable versions each year.
2. Jenkins comes with a wide range of plugins - Jenkins has a wide range of plugins that give a developer a lot of added features and power over the already feature-rich Jenkins installation. These plugins help developers extend Jenkins into their own custom tools for their projects.
3. Jenkins integrates and works with all major tools - Jenkins being the popular tool it is, integrates with all major version control tools like CVS, Subversion, Git, build tools like Apache Ant and Maven and even other tools like Kubernetes and Docker.

4. Jenkins is flexible - With its wide range of plugins and open architecture, Jenkins is very flexible. Teams can use Jenkins in projects of various sizes and complexity.

Jenkins places no limits on the kinds and numbers of servers that can be integrated with it. Teams across the globe can work towards continuous delivery, seamlessly.

Cons of Jenkins:

1. Unpredictable Costs - The costs of hosting the server (which isn't free) that Jenkins runs on cannot be predicted easily. It is not always possible to predict the kind of load (depending on the number of commits, volume of code, volume of assets etc.) that the server is going to serve. Hence the cost factor, even if Jenkins itself is free, remains unpredictable.
2. Lack of Governance - Jenkins' management is generally done by a single user and that leads to tracking and accountability problems with the pushed code. There is tracking that exists on the Version Control Server but there is no tracking of that kind of Jenkins itself, which is a concern.
3. Lack of Analytics - Jenkins doesn't provide any analytics (there are plugins but they are not enough) on the end-to-end deployment cycle. This again goes back to the lack of overall tracking that contributes to the lack of analytics as well.

Results and Discussions:

By following the above steps, we were able to make two improvements to the existing service:

- a. Implement serverless architecture and keep the text extracting microservice as a central service. Earlier it had to be a python script on the user's machine. Every time any enhancement had to be made, a new build had to be deployed. In case the client's IT department did not allow the software to make any kind of external API calls, a server had to be deployed on which the text parsing service had to be deployed, which increased the cost as an extra server had to be hosted.

Now, any change can be made by simply updating the code in the git repository. As soon as the master branch of the code will be updated, the Jenkins Pipeline will be run and the AWS Lambda function that we earlier deployed will be updated.

This helps us move towards FaaS (Function as a Service) architecture.

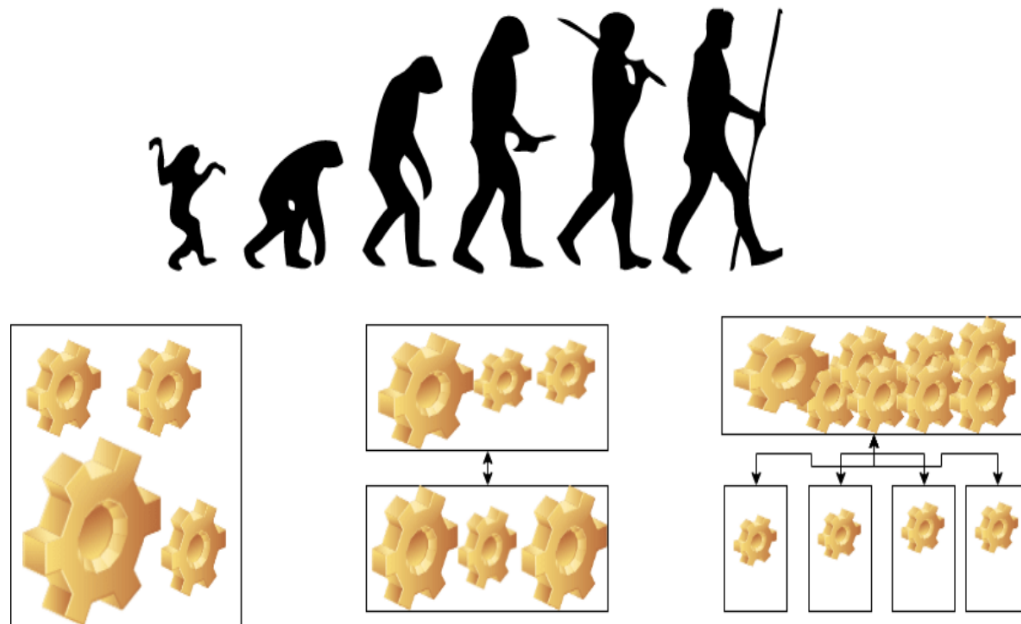


Fig.29: Breaking down the monolithic architecture

- b. By using custom text parsing logic for each EMR, Regular expressions, using a parsing area with a lot greater buffer than before and a configuration driven backend, we were able to improve the text parsing functionality significantly. This allowed us to make an enterprise level solution which, when integrated with a user friendly frontend as a tool/assistant, can help solve a major issue in US Healthcare. It will help save time of care provider by directly showing the important and necessary information about the patient by parsing the screen and searching for the patient in the database while the care provider can interact with the patient. Of the 15 minutes that a patient gets with the

doctor(on an average), 4-5 minutes are wasted in searching for important information. This tool can save these 5 minutes and show the information within seconds of opening the patient's chart in the EMR. This will improve the quality of service provided by the doctor drastically.

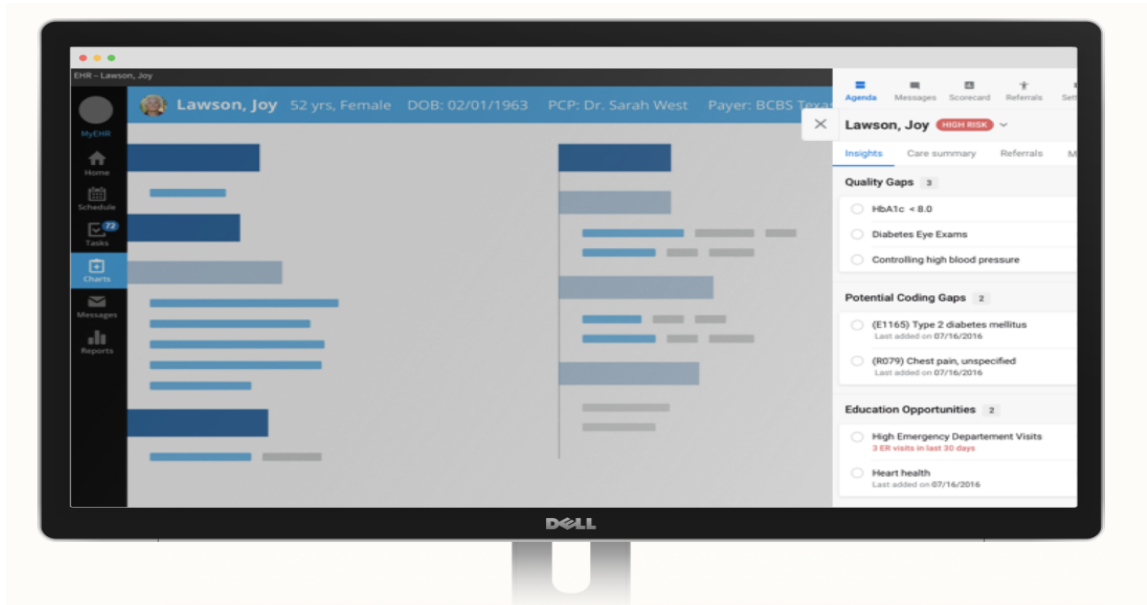


Fig.30: Parsing name from the EMR on the screen

Chapter 5

Future Scope:

The concepts and steps mentioned have tremendous scope in the future.

Serverless architecture can be implemented in the complete backend instead of using just one microservice. This would break down the monolithic backed structure into a number of microservices, each of which will perform only one action. Each microservice can be added as a plugin or disabled depending on the need. This would help boost scalability of the project and reduce efforts and time consumed in deployment, upgradation and fixes.

Human Name parsing can be used in multiple number of fields. It can be used to automate analysing data from forms and images, which would drastically improve the quality of services that we have. Eg: filling e-forms, analysing receipts and bills, documents etc.

Chapter 5

References

1. Image Processing: lijun sun, Structural behaviour of asphalt pavements
2. AWS Lambda and EC2 AMI: Official AWS documentation
<https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
3. Django: Official Django documentation
<https://docs.djangoproject.com/en/2.2/>
4. MongoDB: Official MongoDB documentation
<https://docs.mongodb.com/manual/>
5. Docker: Official Docker documentation
<https://docs.docker.com/engine/docker-overview/>
6. Serverless: Official Serverless documentation
<https://serverless.com/framework/docs/>
7. Introduction to MongoDB
<https://beginnersbook.com/2017/09/introduction-to-mongodb/>
8. Handling Python dependencies in Lambda
<https://serverless.com/blog/serverless-python-packaging/>
9. AWS Lambda with python
<https://stackify.com/aws-lambda-with-python-a-complete-getting-started-guide/>
10. Lambda
<https://hackernoon.com/what-is-aws-lambda-or-serverless-f0a006e9d56c>
11. Instances and AMIs
<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instances-and-amis.html>
12. PyWin32 Documentation
<http://timgolden.me.uk/pywin32-docs/contents.html>

13. Enumerate displays
http://timgolden.me.uk/pywin32-docs/win32api__EnumDisplayDevices_meth.html
14. Enumerate Display Settings
http://timgolden.me.uk/pywin32-docs/win32api__EnumDisplaySettings_meth.html
15. Package to handle multiple Displays
<https://pypi.org/project/Desktopmagic/>
16. Google vision library Docs
<https://cloud.google.com/vision/docs/libraries#client-libraries-install-python>
17. Using Client Libraries
https://cloud.google.com/vision/docs/libraries#using_the_client_library