# Amigo – A Chat-Bot for Cloud Operations Management

A Project Report
Presented to
The Faculty of the College of
Engineering

San Jose State University
In Partial Fulfillment
Of the Requirements for the Degree

**Master of Science in Software Engineering**

By

Swetha Muchukota, Chetan Punekar, Watsh Rajneesh, Ashutosh Sharma

May 2017

**APPROVED**

Andrew Bond, Project Advisor

Dan Harkey, Director, MS Software Engineering

Xiao Su, Department Chair

## ABSTRACT

Amigo – A Chat-Bot for Cloud Operations Management

By Swetha Muchukota, Chetan Punekar, Watsh Rajneesh, Ashutosh Sharma

This project aims at simplifying Cloud Operations Management, triaging workflows and DevOps, in general, using Chat-Bot as the user interface. Instead of interacting with a Web UI, the user can send messages to the Chat-Bot, which will recognize the intent of the messages and translate it to respective service API calls.

Cloud Operations Management today involves a lot of operation workflows that are repetitive and mundane. Automation of those tasks with scripting requires development and maintenance of the script code. Using a Web UI can be slower and using a command line interface can be tedious.

In this project, we propose a Chat-Bot as UX (user experience) for performing cloud operations. A user will type in the intended operations in plain English and Chat-Bot will translate it to the corresponding one or more APIs call(s) to be made on the cloud service. This will alleviate the tedium of remembering the commands on the command-line interface or filling out a data form in a Web UI. Access control will be implemented so only privileged users will be allowed to perform their operations via the Chat-Bot. In a triage session, a user will interact with the bot performing a sequence of operations. All triaging sessions will be recorded automatically in the backend, enabling reporting of the whole triage session and thus building a shared knowledge base over time.

## **Acknowledgements**

The authors are deeply indebted to Professor Andrew Bond and Professor Thomas Hildebrand for their invaluable comments and guidance in the preparation of this project.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1. PROJECT OVERVIEW

## 1.1 INTRODUCTION

**ChatOps** is an approach to communication that allows teams to collaborate and manage many aspects of their infrastructure, code, and data from the comfort and safety of a chat room. It is a chatbot program that is designed to simulate the conversation with human users in the context of IT operations management. It enables improving how IT teams collaborate to handle DevOps by making it more visible, efficient and simple.

Following are the factors justifying the importance of ChatOps:

1. **Collaborative DevOps -** Chat solutions like Slack are highly extensible, have integrations with AWS Lambda, and third party services, such as PagerDuty, Jira, NewRelic, GitHub.
2. **Better and easier than IRC** - Registering to IRC (Internet Relay Chat) services are cumbersome and they lack the flexibility and integrations provided by enterprise messengers like Slack.
3. **Not behind a VPN** - Team collaboration services like Slack and HipChat are hosted as SaaS applications in the cloud and can be accessed by team members without logging in to the VPN.
4. **Makes operations easy from mobile devices** - Messengers like Slack and HipChat have mobile apps which can be used to collaborate from user's mobile phone. Having the ChatOps support on such mobile apps will make it more convenient for teams to collaborate and carry forward the DevOps culture.

5. **Built-in archiving** - The messages typed in the chat are persisted in the SaaS Chat service's DB. This can be used to search the data in previous chat sessions.
6. **Easy to share screenshots, file attachments** - Slack and HipChat make sharing of screenshots and file attachments easy.
7. **Makes incident response across teams much easier to handle -**

- Monitoring solutions like Data Dog for AWS monitoring and PagerDuty have integration with slack.
- GitHub has a chat bot that sends notification message on the slack channel when a new commit is pushed to a branch.
- Jenkins has integration with Slack so users can perform build operations and run the Jenkins jobs right from within Slack
- AWS Cloud Watch events integration with Slack - Cloud Watch events are sent to Slack channel.

### 1.1.1 Proposed Areas of Study and Academic Contribution

This section describes the research work carried out for knowing the state-of-the-art for Chatbot design techniques, Natural language processing (NLP), Artificial intelligence markup language (AIML) and micro services based architecture.

## 1.2 LITERATURE SEARCH

Chabot (Chatter-Bot) is a computer program with some intelligence. It assists humans with the ability to provide/respond with suitable answers to the user's questions by extracting the intent, keywords from the question/message. The input given to this program is a natural language text and the application responds with an answer using its intelligence. This process continues along with the conservation.

Amigo - A Chat-Bot for Cloud Operations Management is an approach to communication that allows teams to collaborate and manage many aspects of their infrastructure, code, and data from the comfort and safety of a chat room. It is a Chabot program that is designed to simulate a conversation with human users in the context of IT operations management. It enables improving how IT teams collaborate to handle DevOps by making it more visible, efficient and simple.

The bot system began in an early nineteenth century where the first chess- playing machine was built such that a system act as a player and another player a human can play against it, such that system giving a feeling of another person existence. The challenge to a Chabot developer here is to build a large knowledge base to make it perfect thereby ensuring acceptable answers to the question.

Following are the areas where the literature survey was focused on for the conceptualization of the project:

- Chatbot Architecture
- Microservice-based Architecture
- Chatbot Design Techniques
- Natural Language Processing
- Artificial Intelligence Markup Language (AIML)
- Pattern Matching

Sameera A. Abdul-Kader and Dr. John Woods (2015) work explains Chabot strategies, fundamental design techniques. Their work has given a good introduction of three important components of Chabot software package:

- Responder – Interface between bot's main routine and user
- Classifier – Responsible for filtering, clustering, normalizing the input into logical components
- Graph Master – responsible for pattern matching

They have suggested the most important techniques needed to start with Chabot. Some of them were:

- Parsing – to analyze and manipulate the text.
- Pattern matching
- AIML – core technique, AIML is base of chatbot brain
- Markov Chain – to build responses that are probabilistically and consequently applicable.

Their work has enlightened us with further areas where literature survey needs to be focused.

Alan Shaw (2014) proposed a framework for parsing sentences from a user during the chat a Chabot using a strategy called 'answer strategy'. The Framework has exposed us to concepts of pattern matching with three simple rules for searching keyword matches in an input sentence.

Abbas Saliimi, Lokman and Jasni Mohamad Zain (2010) explains the One-Match and All-Match Categories (OMAMC) for keywords matching in Chabot. Using this technique there is a noticeable improvement in matching time and matching flexibility.

Md. Shahriare Satu, Md. Hasnat Parvez, Shamim-Al-Mamun (2015) work reviewed AIML integrated applications. Their work has highlighted how AIML (Artificial Intelligence Markup Language) helps to build simple, efficient and easy to configure conversational agent artificially chatbot.

Amittai Axelrod (2015) presented development platform for natural language processing using Amazon Elastic Cloud Service

## 1.3 CURRENT STATE OF THE ART

Killalea, Tom (2016) explains an approach of building distributed system using microservices where there is clear separation and focus on attention to each service. It covers the microservice architectural styles, deployment size, consumer driven contracts.

**Table 1 - Chat-Bot State of the Art**

| S.No. | Paper Title | State-of-the Art Summary | Keywords |
|---|---|---|---|
| **1** | Survey on Chatbot Design Techniques in Speech Conversation Systems | Comparison of different chatbot design techniques | **AIML** **Chatbot** **NLP** **Chatbot Design** |
| **2** | A System of Simple Sentence Parsing Rules To Produce "Answer Matching" Chatbots | Presents a framework to parse user input during a chat | **NLP** **Pattern Matching** |
| **3** | Natural Language Processing Research Using Amazon Web Services | Provides a mechanism for using Amazon Elastic Compute Cloud for natural language processing | **AWS** **NLP** |
| **4** | Review of integrated applications with AIML based chatbot | Brief review of some applications which are used AIML chatbot for their conversational | **AIML Chatbot** |

| | | | |
|---|---|---|---|
| **5** | One-Match and All-Match Categories for Keywords Matching in Chatbot | Proposes a keyword matching technique which improves the performance and | **NLP Pattern Matching** |
| **6** | The Hidden Dividends of Microservices | An approach of building distributed system | **Microservices** |
| **7** | Personaaiml: An architecture developing chatterbots with personality | Flexible architecture that allows the use of different models of personality in the | **AIML Chatbot** |
| **8** | Microservices | Compares microservices to service-oriented Architecture | **Microservices** |
| **9** | AIML Based Voice Enabled Artificial Intelligent Chatterbot | Shows AIML implementation of chatbot | **AIML Chatbot NLP Chatbot Design** |
| **10** | Just. Chat-a platform for processing information to be used in chatbots | A platform, Just. Chat, that helps in the creation of chatbot's knowledge bases | **NLP Chatbot** |
| **11** | The elements of AIML Style | Basics about AIML language | **AIML** |
| **12** | Evolutionary sentence building for chatterbots | Proposes a model in which new sentences can be produced from existing ones using an evolutionary algorithm adapted to the structure of the natural | **AIML Chatbot NLP Chatbot Design** |

| | | | |
|---|---|---|---|
| **13** | **A survey of chatbot system through a Loebner prize competition** | **Explains the different technologies used in the chatbots which have won the Loebner Prize Competition** | **Chatbot** |

# Chapter 2. Project Architecture

## 2.1 INTRODUCTION

Amigo Chatbot project's architecture had the following goals:

1. To support following types of clients –
    a. Enterprise messengers – like Slack or Facebook messenger.
    b. Web UI
    c. Mobile Application
    d. Hardware virtual assistant – like Raspberry Pi based virtual assistant or Amazon Echo.
2. To be highly scalable so several hundreds of users can be served simultaneously.
3. To be very reliable so no user message is dropped or goes un-serviced.
4. To support multiple cloud provider management.
5. To carry out all message exchanges securely.

The following diagram shows the subsystems of the architecture and how they are connected.

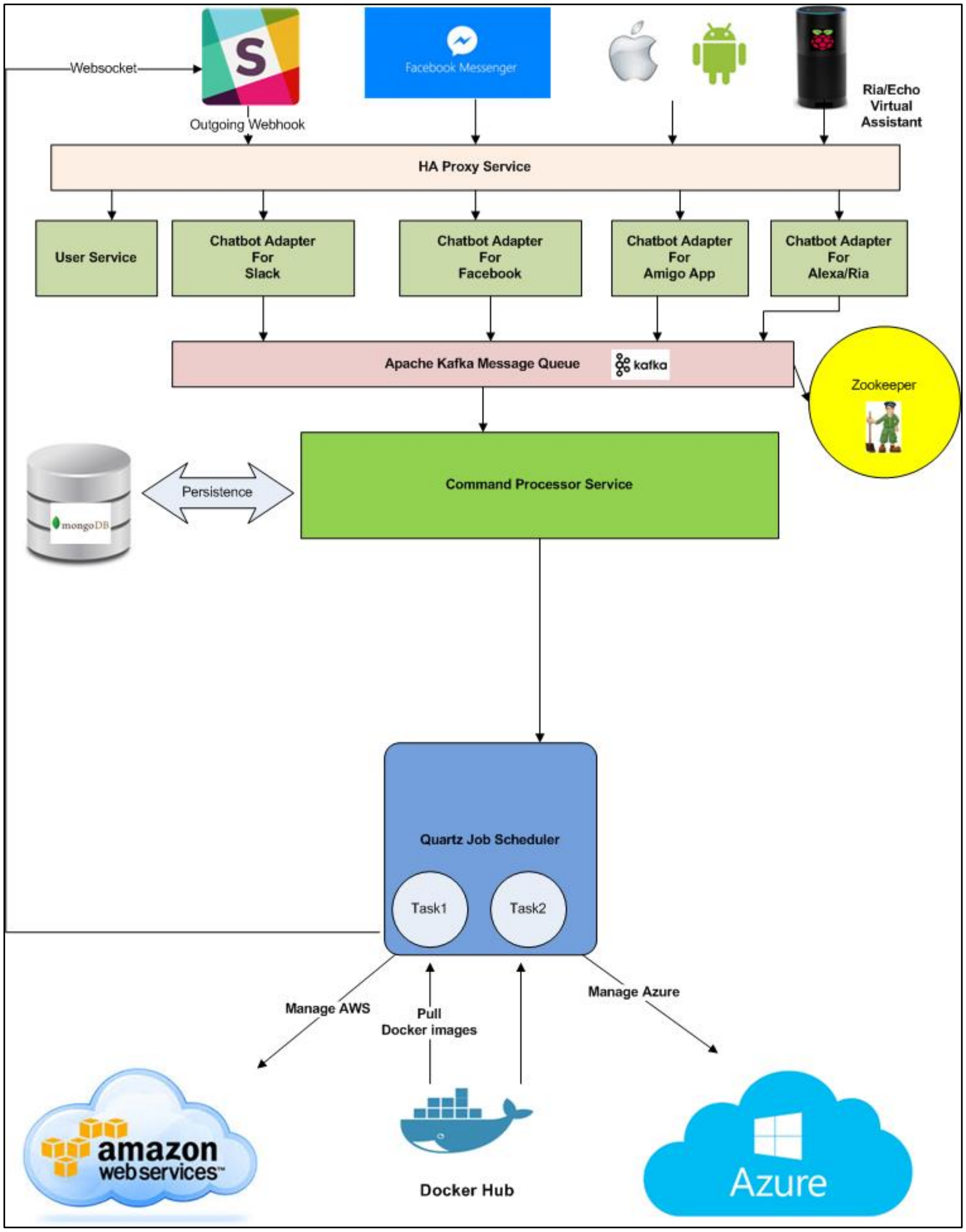


**Figure 1 - Amigo Chatbot Architecture**

The above architecture attempts to meet the goals in the following manner:

1. Multiple client types – Slack Messenger, Web UI, Mobile App and Amazon Echo.
2. Highly scalable to serve hundreds of users concurrently –
    a. Apache Kafka message queue can be scaled to deal with millions of firehose-style events generated in rapid succession. It guarantees low-latency, "at-least-once", delivery of messages to consumers. It supports retention of data for offline consumers, which means that the data can be processed either in real-time or in offline mode.
    b. Use of **Docker Swarm (mode)** container manager to provide auto-scaling and fault tolerance.

The figure below shows a typical Docker Swarm cluster with 3 nodes. Node-1 is the manager and the other 2 nodes are workers. We deploy a service to swarm cluster and swarm cluster manager places the container for the service on one or more of the nodes in the cluster. If a node in the cluster goes down, Swarm will re-distribute the service containers across the remaining nodes thus ensuring that the deployed number of replicas for a service always remains same. So, Swarm starts new service containers in the remaining nodes in the cluster. When there is heavy load on the system the Swarm cluster can be scaled to add more nodes. Services can be scaled horizontally to have more replicas. Services can be updated to scale vertically by adding more resources (say increase memory limit from 200MB to 500MB).

Docker networking can be used to create multiple networks and provide isolation and security to services. For example, as shown in the figure below, App and DB are connected to App-Net overlay network. DB can only be accessed via the App REST endpoints. So other application services will not be connected to the same App-Net network.

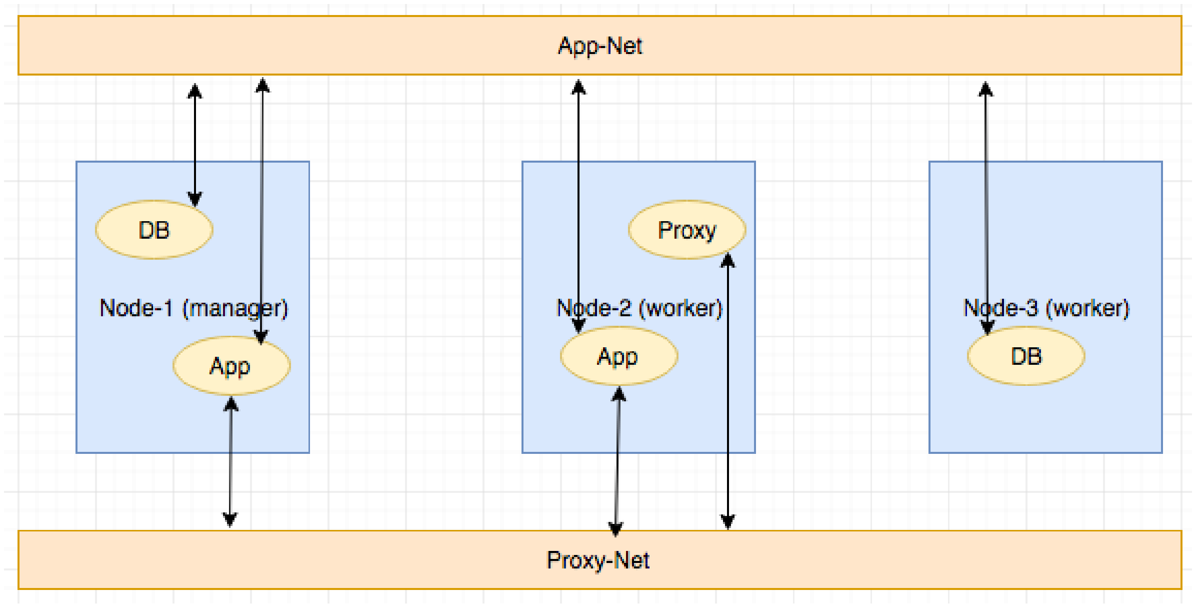


**Figure 2 - Docker Swarm (mode)**

## 2.2 ARCHITECTURE SUBSYSTEMS

Following are the major subsystems in the architecture:

1. **Slack Bot Service** – This service integrates with Slack messenger and listens for all messages sent either directly to the amigo chatbot or sent to a channel in which amigo chatbot was a participant of. Once the message is received, slack bot service will forward it to the Chatbot service for further processing.
2. **Wit.ai Service** – it is a third-party service which is used to process incoming user message and extract the intent from it.
3. **Chabot Service** – use the wit.ai service to parse the message for its intent. Once the intent is found, the message is published to the **user.msg** topic on the Kafka message queue. This service will be invoked by client specific adapter services that receive messages from their respective client types and forward to this service for further processing.
4. **RIA Bot Service –** will receive message from RIA (Raspberry Pi Virtual Assistant) and forward to Chatbot service for further processing.
5. **User Service** – This service will provide REST endpoints for user management. This includes registering new users, authenticate a user, persist a user profile, etc. A user will be required to first sign up with the Amigo Chabot before using the service in an enterprise manager. This is required because during sign-up user can provide their cloud provider credentials, their slack user ID, their virtual

assistant ID to the system. Once signed up user can always login and manage their profile information.

6. **Command Processor Service** – This service will poll for user messages being published to the **user.msg** topic. Once the message is available, one of the consumer threads in the Amigo Chabot consumer's group will process the message.

7. **Apache Kafka Message Queue** – Kafka is highly scalable and a reliable message queue built like a distributed commit log. It can provide a durable record of all transactions that can be played back to recover the state of a system. It provides redundancy, which ensures high availability of data even when one of the servers faces disruption. Multiple event sources can concurrently send data to Kafka clusters, which will reliably get delivered to multiple destinations. [1]

The main reason for choosing Kafka was its easy to scale architecture and it's "at-least once" message delivery guarantee.

8. **Apache Zookeeper Service Discovery** – Kafka uses Zookeeper as the distributed configuration store. It forms the backbone of Kafka cluster that continuously monitors the health of the brokers. When new brokers get added to the cluster, Zookeeper will start utilizing it by creating topics and partitions on it.

9. **Docker Hub** – is a registry for Docker images. It is used to store the cloud provider specific image that pre-installs the cloud provider's CLI client tool so the image

can be used to run a container that can be passed any command that the CLI tool can execute.

10. **Amazon Web Services Public Cloud** – Amazon Web Services is the leading public cloud provider. This is one of the cloud providers that can be managed via the Chabot's conversational interface.
11. **Infrastructure Services** – will provide logging, visualization and monitoring for the application services and the Docker Swarm cluster.

# Chapter 3. TECHNOLOGY DESCRIPTIONS

Amigo Chatbot will be developed using modern agile methodology of software development with microservices based software architecture and deployed on Docker containers. Following table lists all the technologies and tools employed in the project:

| Category | Technology | Description |
|---|---|---|
| **Client Technologies** | React JS [2] | A JavaScript library for building user interfaces |
| | React Native [3] | Building mobile apps with React that work same as native apps and uses same fundamental UI building blocks as regular iOS and Android apps using JavaScript and React. |
| | Slack bot users [4] | Enables teams to conversationally interact with external services or custom code by building bot users. |
| | Raspberry Pi Intelligent Assistant (RIA) [5] | A voice controlled virtual assistant using speech-to-text engines, text-to-speech engines and conversations modules. |
| **Middle Tier Technologies** | Docker Containers [6] | Docker is a tool designed to make it easier to create, deploy and run applications by |

| | | |
|---|---|---|
| | | using containers. Containers allow a developer to package up an application with all the parts it needs, such as libraries and other dependencies, and ship it all out as one package. Docker containers are much more lightweight and use far fewer resources than virtual machines. |
| | Docker Swarm mode [7] | Cluster management and orchestration features embedded in the Docker engine are built using **SwarmKit**. Docker engine participating in a cluster are running in swarm mode. |
| | Docker Hub [8] | Docker Hub is a cloud-based registry service which allows one to link to code repositories, build images and test them, stores manually pushed images, and links to the Docker Cloud so one can deploy images to hosts. It provides a centralized resource for container image discovery, distribution and change management, |

| | | |
|---|---|---|
| | | user and team collaboration and workflow automation throughout the development pipeline. |
| | Wit.AI [9] | Wit.AI makes it easy for developers to build applications and devices to which they can text to or talk to. Wit.AI learns from every interaction from the community and all learning is shared across developers. |
| | Consul [10] | Consul is a tool for discovering and configuring services in the infrastructure. |
| | Apache Kafka [11] | Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, fast and runs in production in thousands of companies. |
| | Apache ZooKeeper [12] | ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization and providing group services. |

| | | |
|---|---|---|
| | HA Proxy [13] | HA Proxy (High Availability Proxy) is an open-source load-balancer which can load balance any TCP service. It is free, very fast and reliable solution that offers load balancing, high availability and proxying for TCP and HTTP-based applications. |
| | Elastic Search [14] | Elastic search is a highly-scalable open-source full-text search and analytics engine. It allows you to store, search, and analyze big volumes of data quickly and in near real time. |
| | Logstash [15] | Logstash is part of the Elastic Stack along with Beats, Elasticsearch and Kibana.<br>It is used to collect, aggregate, and parse your data, and then have Logstash feed this data into Elasticsearch. |
| | Kibana [16] | Kibana is an open source analytics and visualization platform designed to work with Elasticsearch. Kibana is used to search, view, and interact with data |

| | | |
|---|---|---|
| | | stored in the Elasticsearch indices. One can easily perform advanced data analytics and visualize data in a variety of charts, tables and maps. |
| | Logspout [17] | Logspout is a log router for Docker containers that runs inside Docker. It attaches to all containers on a host, then routes their logs wherever you want. It also has an extensible module system. It’s a mostly stateless log appliance. It captures container logs written to stdout and stderr. |
| | Prometheus [18] | Prometheus is an open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach. |
| | Grafana [19] | Grafana is open source software for time series analytics and visualization. |
| | Node-Exporter [20] | Node exporter is exporter of hardware and OS metrics exposed by *NIX kernels |

| | | |
|---|---|---|
| | | to Prometheus. It is designed to monitor the host system. |
| | Google cAdvisor [21] | cAdvisor (container advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. |
| | Dropwizard Java Microservices framework [22] | Dropwizard is a Java framework for developing ops-friendly, high performance, RESTful webservices. Dropwizard pulls together stable, mature libraries from the Java ecosystem into a simple light-weight package. It has out-of-the-box support for configuration, application metrics, logging, operational tools and much more. |
| | Quartz Job Scheduler [23] | Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application. It can be used to create simple or complex schedules for |

| | | |
|---|---|---|
| | | executing tens of thousands of jobs; jobs whose tasks are defined as standard Java components that may execute virtually any task. It includes many enterprise class features like support for JTA transactions and clustering. |
| | Jersey JAX-RS for REST API Client and Service [24] | Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation. |
| **Data Tier** | Mongo DB [25] | Mongo DB is a document oriented NoSQL database that provided high performance, high availability and automatic scaling. |

## 3.1 CLIENT TECHNOLOGIES

Amigo Chat-bot will integrate with enterprise messengers like **Slack Messenger**.

Web UI for Chat-bot where user profile will be created and login will be performed. For Web UI, we will use HTML5, CSS3.0, React.js and Twitter bootstrap. **React.js** is an open source JavaScript library which provides view to render as HTML. React is declarative, component based library which makes it easy to create interactive UI. It only changes the component when the state changes. It is efficient as we can reuse the component.

For mobile application, **React Native** will be used to build cross platform mobile UI.

**Virtual Assistant** integration for voice based Cloud Ops.

### 3.1.1 REACT JS

React JS is an open source framework for web UI development from Facebook. In the project, it is used to design the web UI for user registration.

### 3.1.2 REACT NATIVE

React Native is used for the cross platform mobile UI implementation.

### 3.1.3 SLACK BOT USERS

Slack Bot users are very like slack human users in that they too have a profile, a name and can be part of a team or receive and send private message. The one difference is that bot users are controlled programmatically via bot user token that accesses one or more slack

API. The have limited set of slack APIs access compared to human users. Also they don’t need to authenticate or login to slack.

There are 2 kinds of bot users:

1. Custom Bots
2. App Bots

### 3.1.3.1 CUSTOM BOT USERS

These are the custom bots that are local to a team. When the intent is to build a bot specific to the needs of one slack team and not to build a bot which can be shared across teams then a team member of a slack team can build a custom bot user.

### 3.1.3.2 APP BOT USERS

When the intent is to distribute the bot user to other teams then it can be achieved by attaching bot user with a slack app. The bot user presentation and functionality can be controlled by the bot owner even after it has been installed. Slack App functionalities like incoming web hooks and slash commands are also part of the distribution.

In the project, we use App Bot users as the intent is to share the Amigo slack bot user with other teams so they can perform cloud ops management.

### 3.1.4 Raspberry Pi Intelligent Assistant

Raspberry Pi Intelligent Assistant [26] is a hardware voice assistant that is used to interact with the cloud hosted Chatbot service and in turn perform cloud operations management just by talking to the bot.

## 3.2 MIDDLE-TIER TECHNOLOGIES

Middle tier where all the business logic is executed message coming from slack or web application goes to AWS lambda where it passes the message to wit.ai for extracting pattern. If the intent is found then it is pushed to Kafka queue from there it is sent to command processor which will check if there is any intent in the database if so, it will pull the corresponding Docker image to run the command. **Java** micro service framework Dropwizard is used to build some of the services.

### 3.2.1 Docker Containers

**Docker** containers will be used to package and deploy the micro services. Docker hub will be used to pull the aws-cli image to further run the AWS command on it.

### 3.2.2 Docker Swarm Mode

Docker Swarm will be used to manage the cluster of Docker containers across hosts or VMs. All services execute as containers within the nodes of Docker swarm cluster. Docker swarm takes care of the placement of the service containers to the right node. In case a node fails in the cluster or a new node is added to the cluster the service containers are re-distributed across remaining swarm cluster nodes.

### 3.2.3 Docker Hub

Docker Hub is a registry for Docker images. It will host the Docker images for the cloud providers' clients (for example, sjsucohort6/docker_awscli is a Docker image with awscli tool installed for the AWS cloud provider) with which the chatbot service will perform the control and monitoring actions on the cloud providers for the users.

### 3.2.4 Wit.AI

Wit.AI service is used for the purposes of extracting intent from the incoming messages.

### 3.2.5 Consul

Consul service registry is built-in into the Docker engine swarm (mode). So, no separate registry container is required to be installed.

### 3.2.6 Apache Kafka

**Apache Kafka** message queue for storing user messages from Chat-bot micro service to the command processor micro service. It is fast, scalable and fault tolerant which is important aspect of our project. Apache Kafka deals with millions of transactions.

### 3.2.7 Apache ZooKeeper

ZooKeeper is used as a service registry for the Apache Kafka containers only.

### 3.2.8 HA Proxy

This serves as API Gateway in the project and proxies for all incoming requests and routes those requests to the right service container based on the URL pattern. For example, /users messages are routed to user service.

In this project, a variant of HA Proxy is used which is called Docker Flow Proxy [27]. In the project, Docker Swarm Listener [28] is used to auto configure the proxy service based on the configuration information provided to it by each service.

### 3.2.9 Elastic Search

Elasticsearch is used as data store for service logs. Elasticsearch is the search and analysis system. It is the place where your data is finally stored, from where it is fetched, and is responsible for providing all the search and analysis results.

### 3.2.10 Logstash

Logstash, which is in the front, is responsible for giving structure to your data (like parsing unstructured logs) and sending it to Elasticsearch.

### 3.2.11 Kibana

Kibana allows you to build pretty graphs and dashboards to help understand the data so you don't have to work with the raw data Elasticsearch returns.

### 3.2.12 Logspout

Logspout service collects all service logs logged on stdout and stderr and forwards it to Logstash service.

### 3.2.13 Prometheus

Prometheus is used in the project for monitoring and time series data storage. It has the following features:

1. Dimensional data – implements a highly dimensional data model. Time series are identified by a metric name and a set of key-value pairs.
2. Powerful queries – has a flexible query language that allows slicing and dicing of collected time series data to generate graphs, tables and alerts.
3. Great visualization – multiple modes for visualizing data, graph, expression browser and console template language.
4. Efficient storage – of time series in memory and on local disk in a custom format. Scaling can be via functional sharding and federation.

5. Simple operation – each server is independent for reliability, relying only on local storage.
6. Precise alerting – Alerts are defined based on the flexible query language and maintain dimensional information.

### 3.2.14 Grafana

Grafana is used to create dashboards with relevant tables and graphs for the monitoring data in Prometheus data store.

### 3.2.15 Node Exporter

Node exporter is used to export hardware and OS metrics exposed by *NIX kernels to Prometheus. In the project, it is used to collect stats on each swarm node in the cluster and runs as a container within each node (global mode) of the swarm cluster.

### 3.2.16 Google cAdvisor

cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. This data is exported by container and machine-wide.

cAdvisor has native support for Docker containers.

It also runs as a service container on any one of the swarm nodes in the cluster.

### 3.2.17 Dropwizard Java Microservices Framework

Dropwizard is lightweight Java microservice framework. It bundles following open source tools and libraries to create a framework:

1. Jackson – a JSON library
2. Jetty – an embedded web server
3. Jersey – JAX-RS 2.0 reference implementation library

In the project Dropwizard has been used to write all microservices with REST endpoints. This includes the following services:

1. user-service
2. command-processor-service
3. chatbot-service

### 3.2.18 Quartz Job Scheduler

Quartz job scheduler is used to asynchronously perform a task. In the project, it is used to submit a job that performs a series of steps to fulfil the cloud operations management task as desired by the user. At the end of the task the response from cloud provider is stored in the database.

### 3.2.19 Jersey RESTful Web Services Framework

Jersey is indirectly used via the Dropwizard framework. It is a reference implementation for JAX-RS 2.0 API.

## 3.3 DATA-TIER TECHNOLOGIES

### 3.3.1 Mongo DB

**Mongo DB** – A document oriented NoSQL DB that is scalable and easy to use. It guarantees atomicity at document level. In the project, MongoDB is used for the following 2 cases:

1. **User DB** – to store the user profile data. User service provides REST API interface for other services to interact with the user data. User DB is a private datastore for user service.
2. **Command DB** – is used at the command processor microservice to store incoming messages and their responses and to store the user profile data at the user

microservice. Data stored in the command DB is also used in message de-duplication at command processor service.

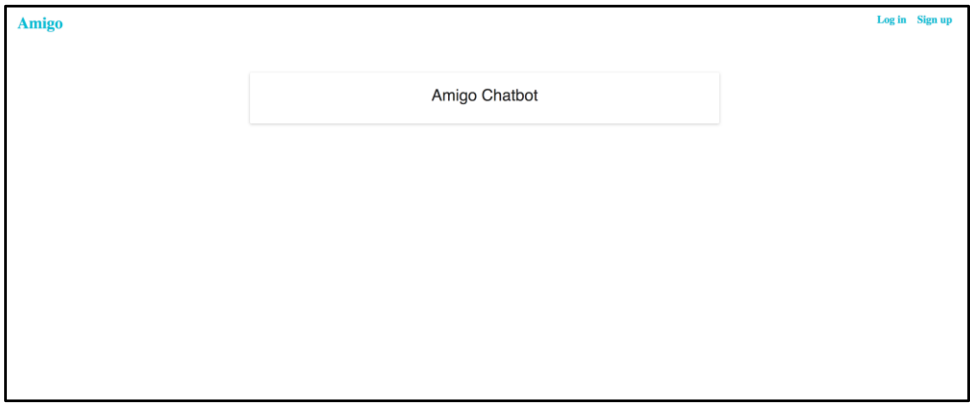
# Chapter 4. Project Design

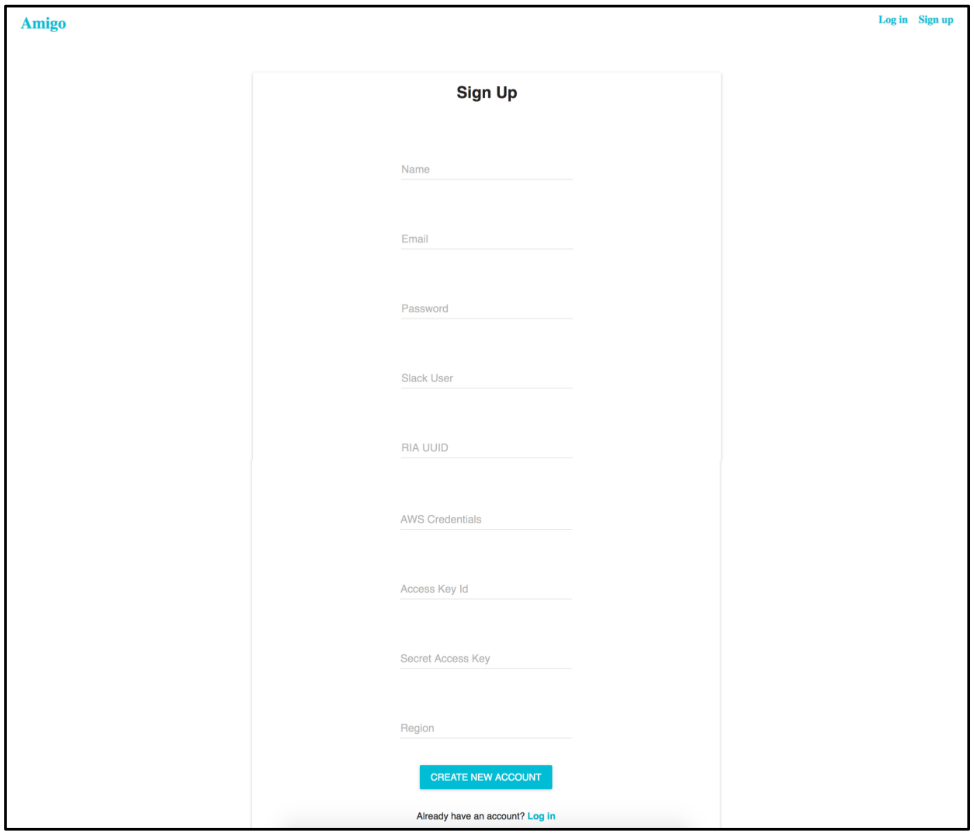
This section talks about the project design objectives and the design patterns employed to attain those goals.

## 4.1 CLIENT DESIGN

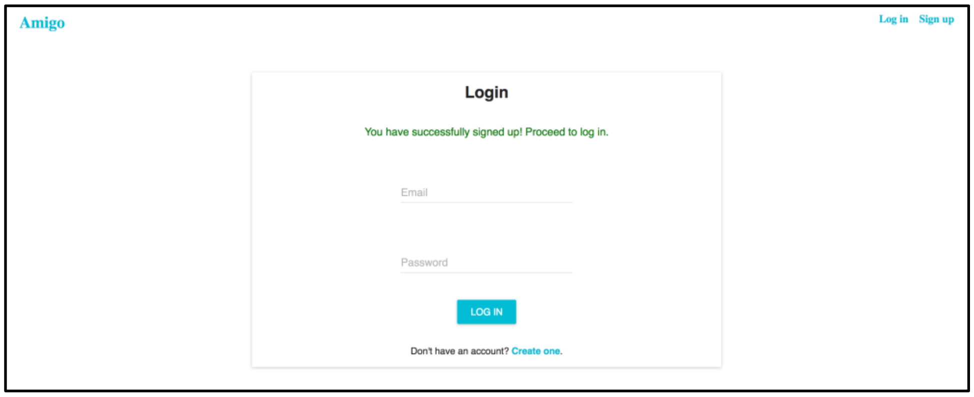
### 4.1.1 Web UI Design

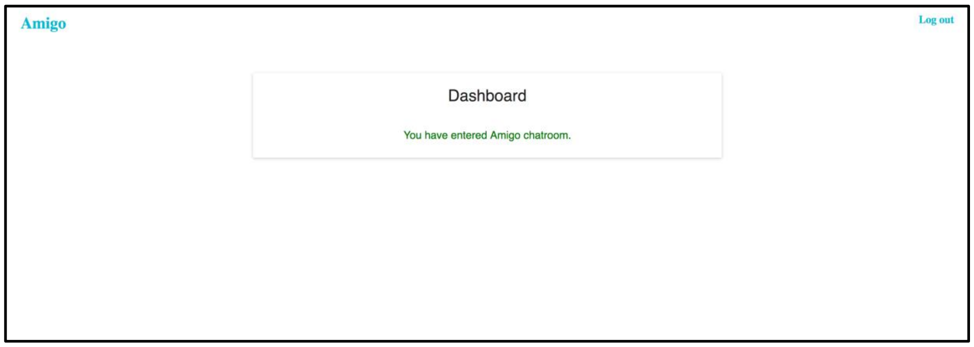
The user should sign up on the Web UI before requesting for Amigo services, following are the screenshots:





Once the user has signed up, he/she can login to the Amigo, following are the screenshots:


Amigo
Log in
Sign up
Login
You have successfully signed up! Proceed to log in.
Email
Password
LOG IN
Don't have an account? Create one.


Amigo

Log out

Dashboard

You have entered Amigo chatroom.

## 4.1.2 Mobile UI Design

### 4.1.2.1 WIREFRAME

Below is the wireframe for our mobile application. It consists of UI elements and the layout of the application that provides user to create account and login to access amigo services. A user can interact with Amigo chatbot to run any cloud related task. Also, user can update their profile.

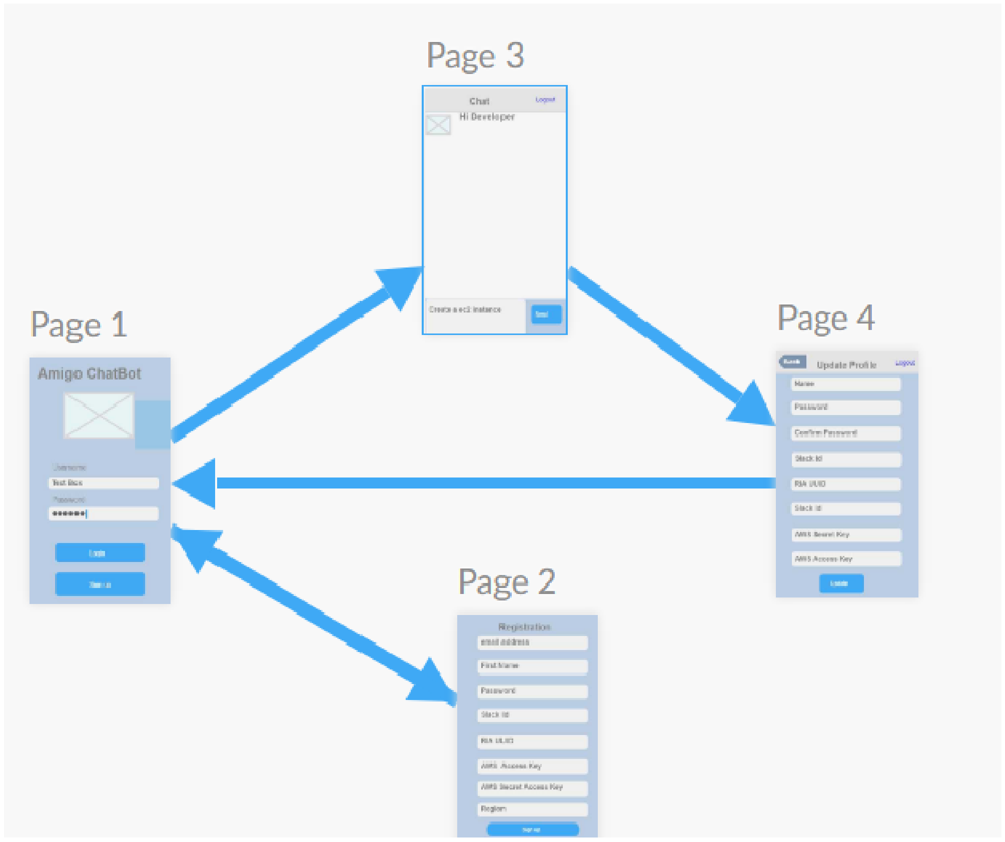


Figure 3 - Mobile UI Wireframe Design

### 4.1.2.2 LOGIN AND REGISTRATION

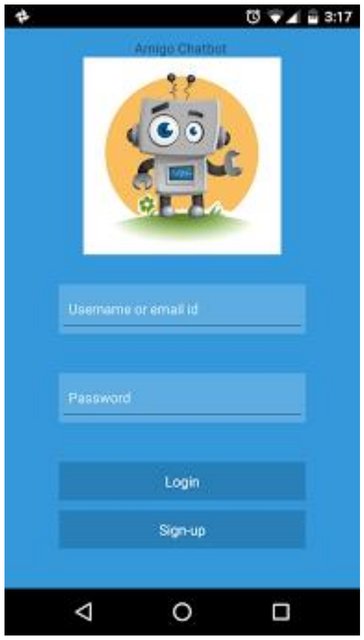


**Figure 4 - Mobile UI Login Screen**

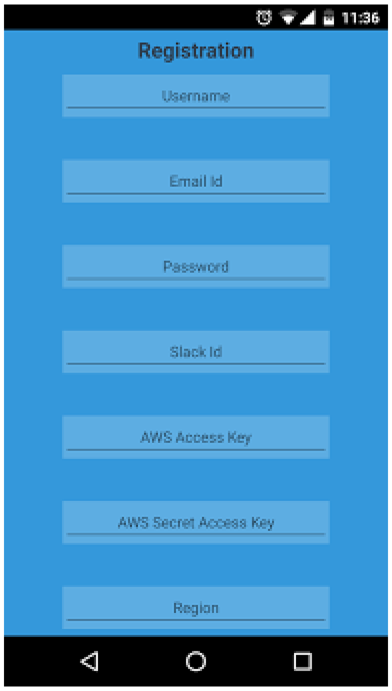


**Figure 5 - Mobile UI Registration**

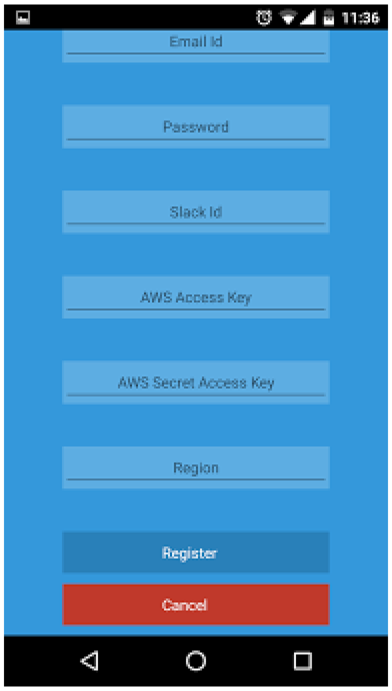
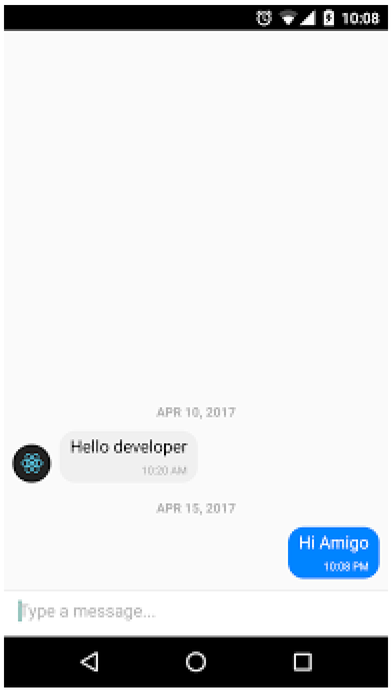


**Figure 6 - Mobile UI Registration II**

#### 4.1.2.3 CHAT UI

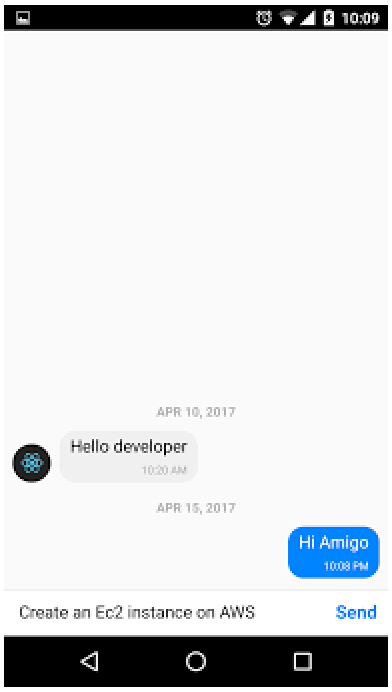
UI design for chat messenger. User can interact with amigo through this interface. We are using react-native-gifted-messenger for the implementation of chat messenger.



Hello developer
10:20 AM

Hi Amigo
10:08 PM

**Figure 7 - Mobile UI Chat**



Hello developer
10:20 AM

Hi Amigo
10:08 PM

Create an Ec2 instance on AWS

**Figure 8 – Mobile UI Chat II**

### 4.1.3 Raspberry Pi Intelligent Assistant Design

. It is made following the nice book on the subject by Tanay Pant - Building a Virtual Assistant for Raspberry Pi (APress). Ria uses Google STT (Speech to text) API and espeak on Linux (or say on OSX) for TTS (Text to speech).

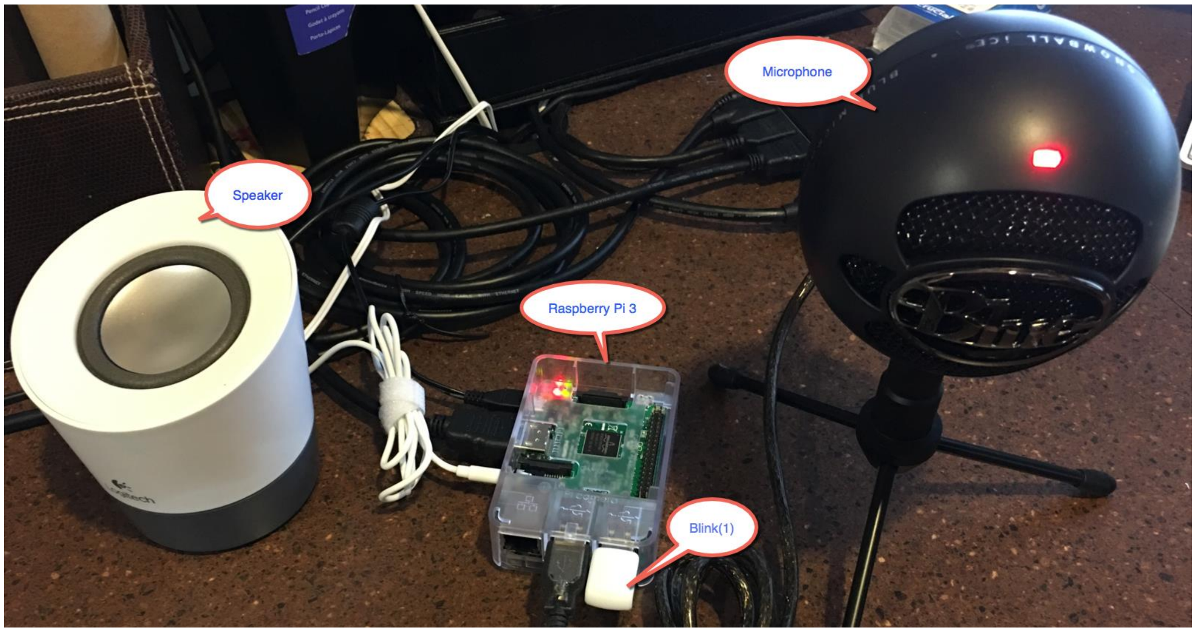


**Figure 9 - Raspberry Pi Intelligent Voice Assistant (RIA)**

The above figure illustrates the Raspberry Pi 3 connected to:

1. Speaker
2. Microphone
3. Blink (1) [29]

Speaker and microphone are attached to the Raspberry pi so the voice assistant can listen and talk to user.

Blink (1) is a small USB light to give to give glance able notice of anything on one's computer or the internet. It makes it easy to connect data sources in the cloud or on computer to a full-color RGB LED so one can know what's happening without checking any windows or going to any websites or typing any command. It has 3 dimensions of information:

1. Color
2. Brightness and
3. Pattern

When user starts talking to RIA the wake-up word is "RIA". At that point, the blink (1) will glow white and flash giving a perception of actively listening to the user.

Once the text has been accepted, the command will be sent to the backend RIA Bot Service and once the result is received from the backend, the white flashing will stop and it will either show a RED (for failure) or GREEN (for success).

## 4.2 MICROSERVICES DESIGN

Building complex applications is inherently difficult. A Monolithic architecture only makes sense for simple, lightweight applications. We will end up in a world of pain if we use it for complex applications.

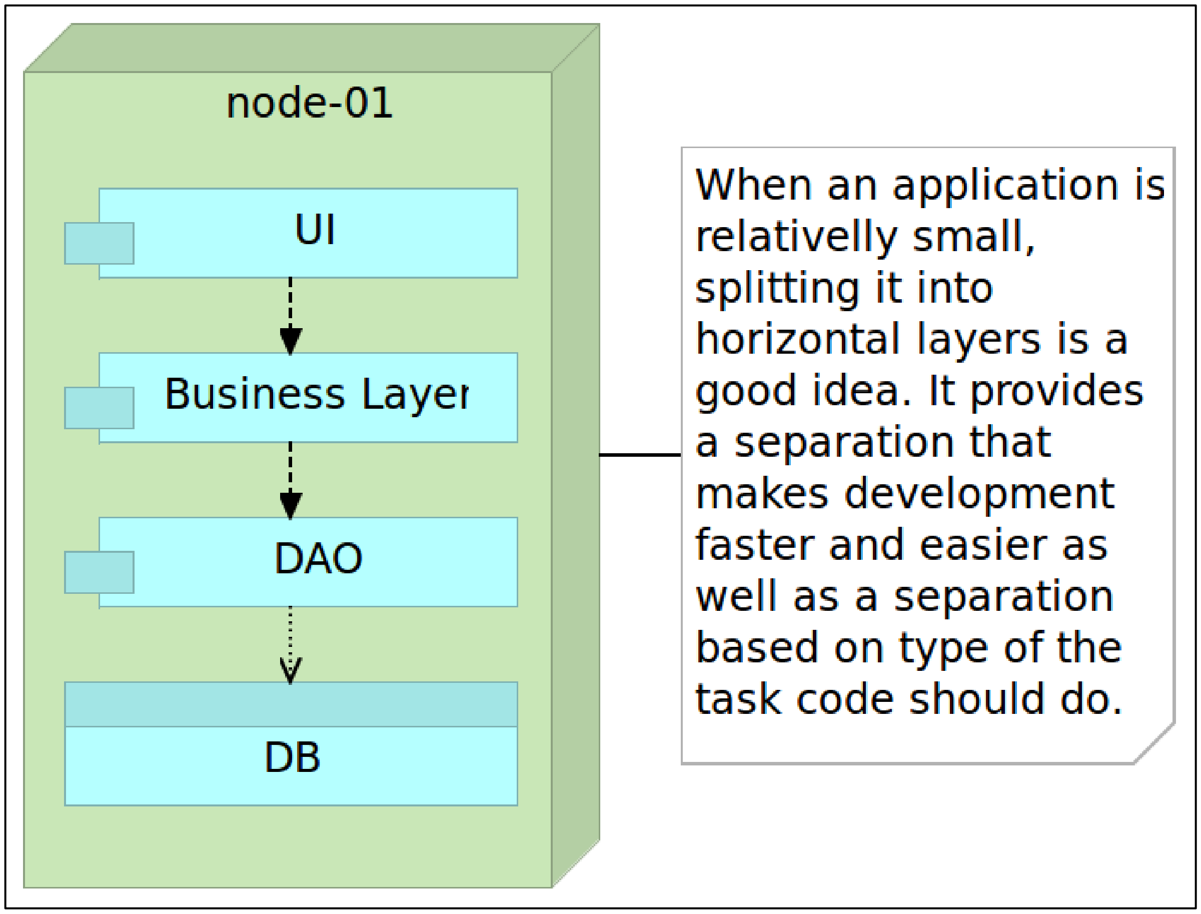

node-01
UI
Business Layer
DAO
DB
When an application is relativelly small, splitting it into horizontal layers is a good idea. It provides a separation that makes development faster and easier as well as a separation based on type of the task code should do.

**Figure 10 - Monolithic architecture good for small applications [30]**

As complexity keeps increasing the monolithic application based on layered design becomes less efficient. With increasing complexity of the application, the speed to develop (to add a new feature), test and deploy the feature keeps decreasing. The larger the code base gets the more time a developer needs to spend in development since number of dependencies increase. More time is spent testing too due to increased complexity.

. Architecture is the key to implementation of most extreme programming practices like continuous integration, test-driven development (TDD), short development cycles, and so on.

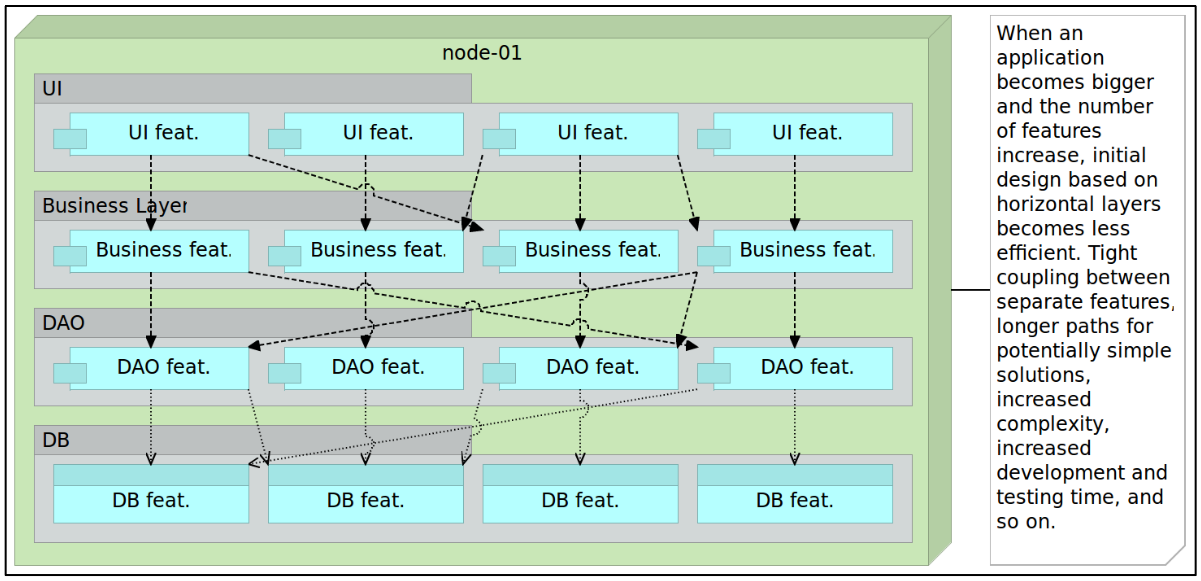

node-01
UI
UI feat.
UI feat.
UI feat.
UI feat.
Business Layer
Business feat.
Business feat.
Business feat.
Business feat.
DAO
DAO feat.
DAO feat.
DAO feat.
DAO feat.
DB
DB feat.
DB feat.
DB feat.
DB feat.
When an application becomes bigger and the number of features increase, initial design based on horizontal layers becomes less efficient. Tight coupling between separate features, longer paths for potentially simple solutions, increased complexity, increased development and testing time, and so on.

**Figure 11 - Monolithic applications tend to become less efficient with time [30]**

Scaling of monolithic applications is very resource inefficient as everything needs to be duplicated.

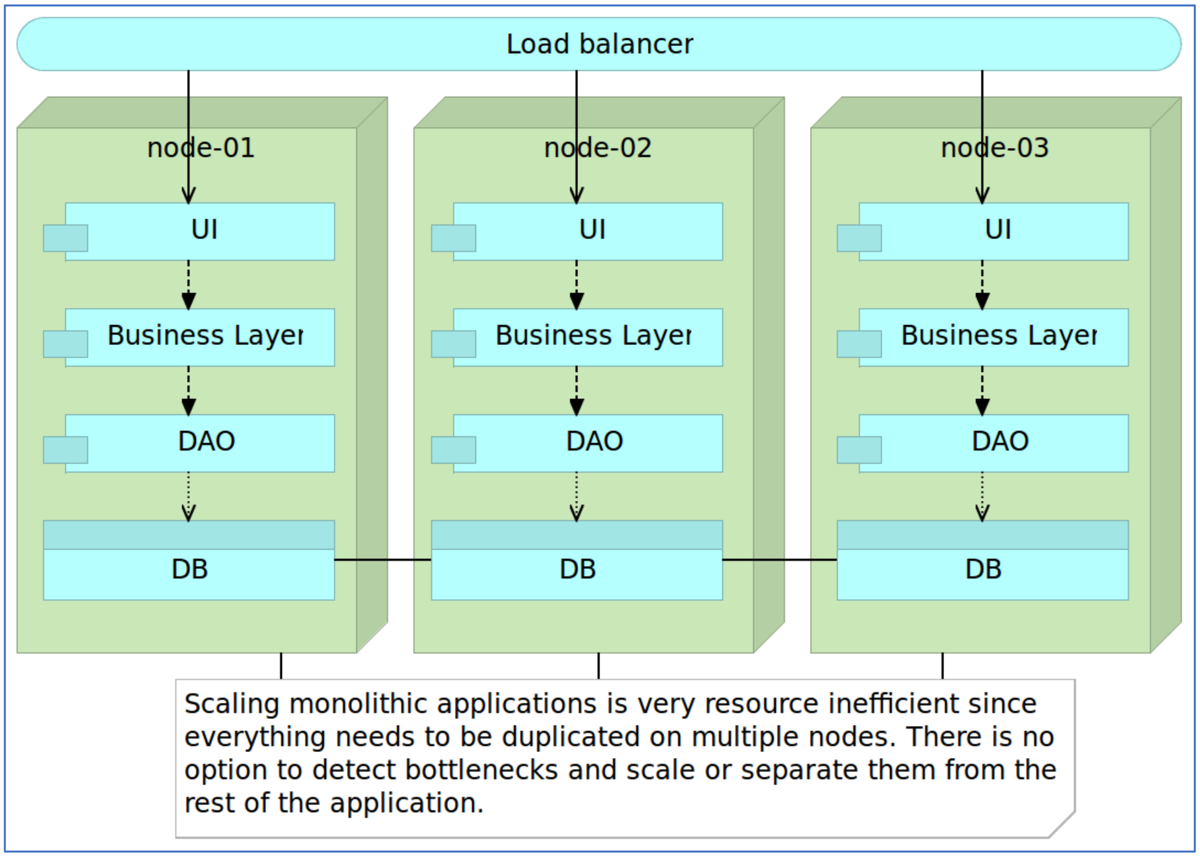

Load balancer
node-01
node-02
node-03
UI
Business Layer
DAO
DB
Scaling monolithic applications is very resource inefficient since everything needs to be duplicated on multiple nodes. There is no option to detect bottlenecks and scale or separate them from the rest of the application.

**Figure 12 - Scaling monolithic applications**

The Microservices architecture pattern is the better choice for complex, evolving applications despite the drawbacks and implementation challenges. The advantages of microservices seem strong enough to have convinced some big enterprise players – like Amazon, Netflix and eBay – to begin their transitions. As opposed to more monolithic design structures, microservices: [31]

1. Improve fault isolation: larger applications can remain largely unaffected by the failure of a single module.
2. Eliminates long-term commitment to a single technology stack: if you want to try out a new technology stack on an individual service, it is easy to do so with microservice

architecture as each service could be written using technology stack that is most appropriate for the task. Dependency concerns will be far lighter than with monolithic designs, and rolling back changes much easier. The less code in play, the more flexible you can be.

3. Makes it easier for a new developer to understand the functionality of a service.

Key aspects of microservices are:

- They do one thing or are responsible for one functionality.
- Each microservice can be built by any set of tools or languages since each is independent from others.
- They are truly loosely coupled since each microservice is physically separated from others.
- Relative independence between different teams developing different microservices (if APIs they expose are defined in advance).
- Easier testing and continuous delivery or deployment

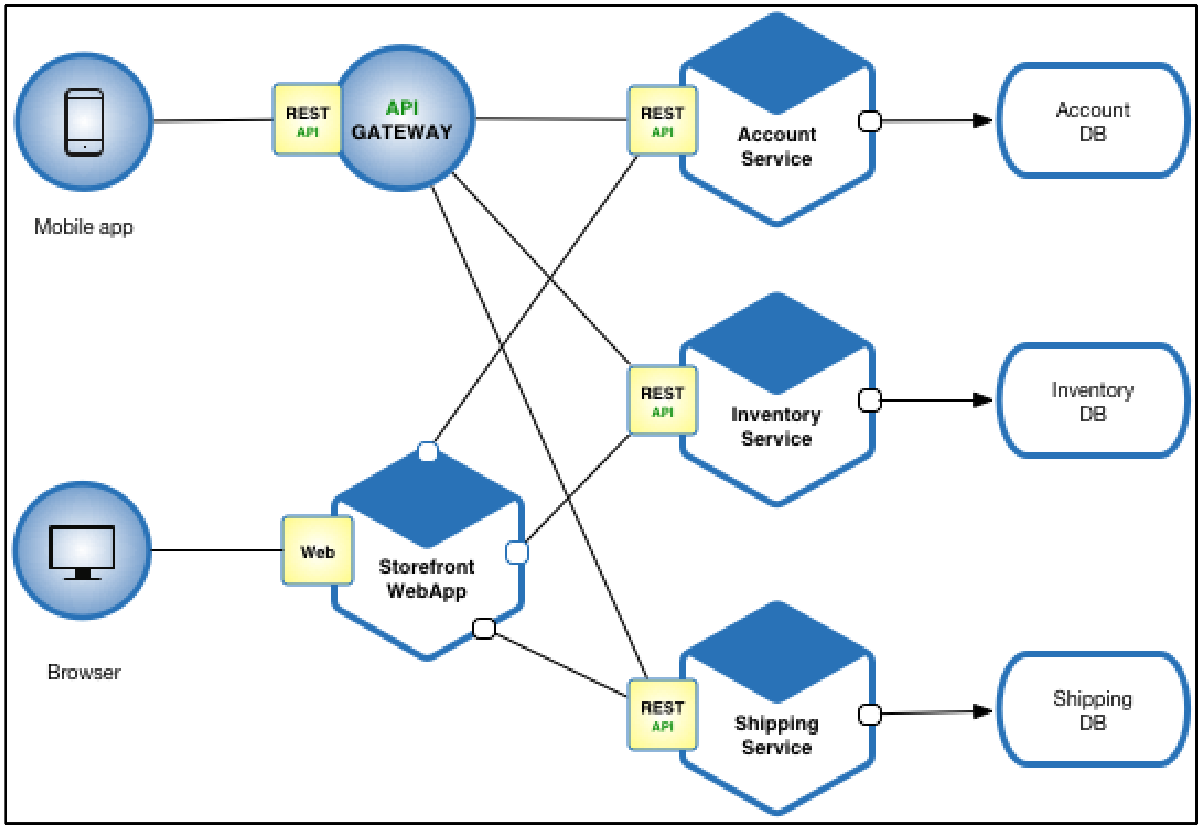


**Figure 13 - A typical microservice application [32]**

Table below lists the advantages and disadvantages of microservice architecture:

| **Architectures** | **Advantages** | **Disadvantages** |
|---|---|---|
| **Monolithic servers** | 1. Good for small applications | 1. Increase in complexity or feature sets increase the amount of code and the time |

| | | |
|---|---|---|
| | | required to develop a feature and test it.<br>2. Scaling is resource inefficient.<br>3. Technology stack once chosen in the beginning limits the choice of technology stack for development of new features. |
| **Microservices** | 1. Easy to scale and more resource efficient. Only scale the microservices that really need scaling.<br>2. We can choose different technology stacks across services within the same application thus selecting the best solution for each service. | 1. Increased operational and deployment complexity.<br>2. Reduced performance due to remote procedure calls between services. |

| | | |
|---|---|---|
| | 3. There is much less code to go through to see what a microservice does. IDEs work faster that way, builds happen in less time, tests complete sooner and overall it speeds up the development time for a new feature.<br>4. Deployment is much faster and easier. Rollback is also fast. If there is an issue with the service the fault in that service is isolated to just that service and other services can continue to work.<br>5. There is no long-term commitment to a technology stack. | |

Weighing the above pros and cons of the 2 architectures, it was evident that microservices architecture is more suitable for the project.

## 4.2.1 Microservices Best Practices

Following are the best practices that need to be applied to a microservices architecture:

### 4.2.1.1 CONTAINERS

Due to the number of microservices could be large for an application it can easily become a very complex endeavor. Each service can be written in a different programming language and can require a different server or can use a different set of libraries. If each service is packaged as a container then most of those problems will go away. All we should do is run the container and trust that everything needed is inside it.

In this project, we used Docker to package the microservices into containers.

### 4.2.1.2 REVERSE PROXY/ API GATEWAY

For most Microservices based applications, it makes sense to implement a Reverse proxy service, which acts as a single-entry point into a system. If there isn't some type of orchestration, dependency between the consumer and microservices becomes so big that it might remove freedom that microservices are supposed to give us. The reverse proxy service is responsible for request routing, composition, and protocol translation.

It provides each of the application's clients with a custom API. The API Gateway can also mask failures in the backend services by returning cached or default data. Their goal is to invoke different microservices and return an aggregated service. They should not contain any logic but simply group several responses together and respond with aggregated data to the consumer.

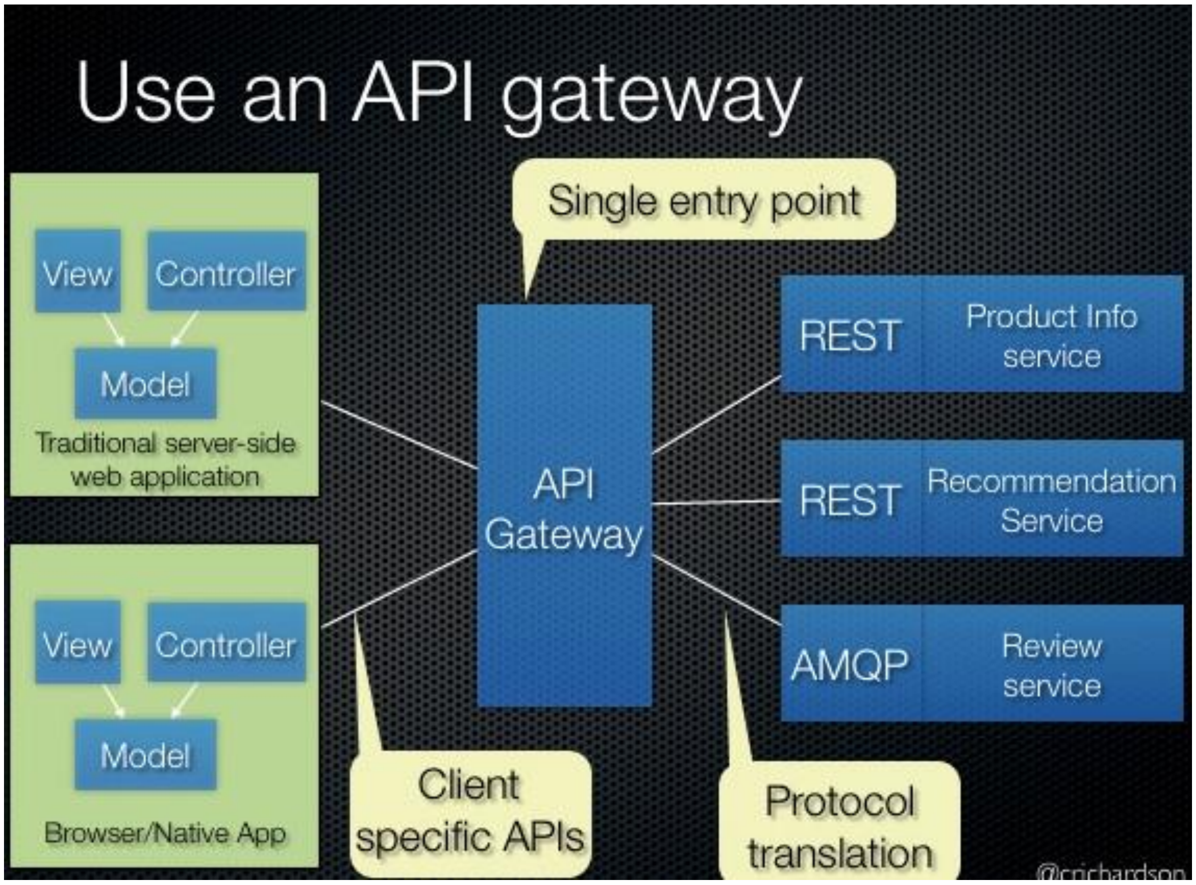


**Figure 14 - API Gateway [33]**

In this project, we employ Docker flow proxy (which is based on Apache HA Proxy) as the API Gateway. All external requests come via the reverse proxy and never directly to a microservice.

#### 4.2.1.4 MINIMALISTIC APPROACH

. Microservices should contain only packages, libraries and frameworks that they really need. The smaller they are, the better.

For each service, we use the following:

1. **Alpine Linux OS** – it is a security-oriented, lightweight Linux distribution based on the musl libc and busybox.
2. **Dropwizard/Jetty** – embedded webserver.

### 4.2.2 MICROSERVICES DESIGN PATTERNS

This section describes the microservices design patterns employed in this project.

#### 4.2.2.1 DECOMPOSE BY SUBDOMAIN

Define services corresponding to Domain-Driven Design (DDD) subdomains. DDD refers to the application's problem space - the business - as the domain. A domain consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

In this project, we have defined the following microservices based on the domains:

1. User-Service: User registration subdomain
2. Slackbot-Service: Slack client adapter
3. Chatbot-Service: Generic client adapter
4. RiaBot-Service: Virtual assistant adapter

5. Command-Processor-Service: Cloud Ops Management action executor

### 4.2.2.2 SERVICE DEPLOYMENT PLATFORM

Use a deployment platform, which is automated infrastructure for application deployment. It provides a service abstraction, which is a named, set of highly available (e.g. load balanced) service instances. The service deployment platform should meet the following requirements:

1. Multiple services could be deployed.
2. Service instances are isolated from one another.
3. Quickly build and deploy a service.
4. Able to constrain the resources (CPU and memory) consumed by a service.
5. Able to monitor the behavior of each service instance.
6. Deployment should be reliable. The desired number of replicas for a service instance should be maintained even in case of failures.

In this project, we use **Docker Swarm** as the service deployment platform.

### 4.2.2.3 MICROSERVICE CHASSIS

Build your microservices using a microservice chassis framework, which handles cross-cutting concerns such as externalized configuration, logging, health checks, metrics, service registration and discovery, circuit breakers.

In this project, we use **Dropwizard** microservice framework as the chassis framework that bundles several libraries together to handle the cross-cutting concerns for the service.

### 4.2.2.4 INTER-PROCESS COMMUNICATION MECHANISM

Microservices must communicate using an inter-process communication mechanism. When designing how services will communicate, we need to consider various issues:

1. How services interact,
2. How to specify the API for each service,
3. How to evolve the APIs, and
4. How to handle partial failure.

There are 2 kinds of IPC mechanisms that Microservices can use,

1. asynchronous messaging and
2. synchronous request/response.

In our design, we employed **Apache Kafka** for asynchronous messaging and REST API calls from one service to another for synchronous request/response.

### 4.2.2.5 SERVICE DISCOVERY MECHANISM

In a microservice application, the set of running service instances changes dynamically. Instances have dynamically assigned network locations. Consequently, for a client to make a request to a service it must use a service-discovery mechanism.

A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and deregistered from the service registry using the management API. The query API is used by the system components to discover available service instances.

There are 2 main service discovery patterns:

1. Client-side discovery
2. Server-side discovery

In client-side discovery, clients query the service registry, select an available instance, and make a request.

In systems that use server-side discovery, clients make requests via a router (API gateway), which queries the service registry and forwards the request to an available instance.

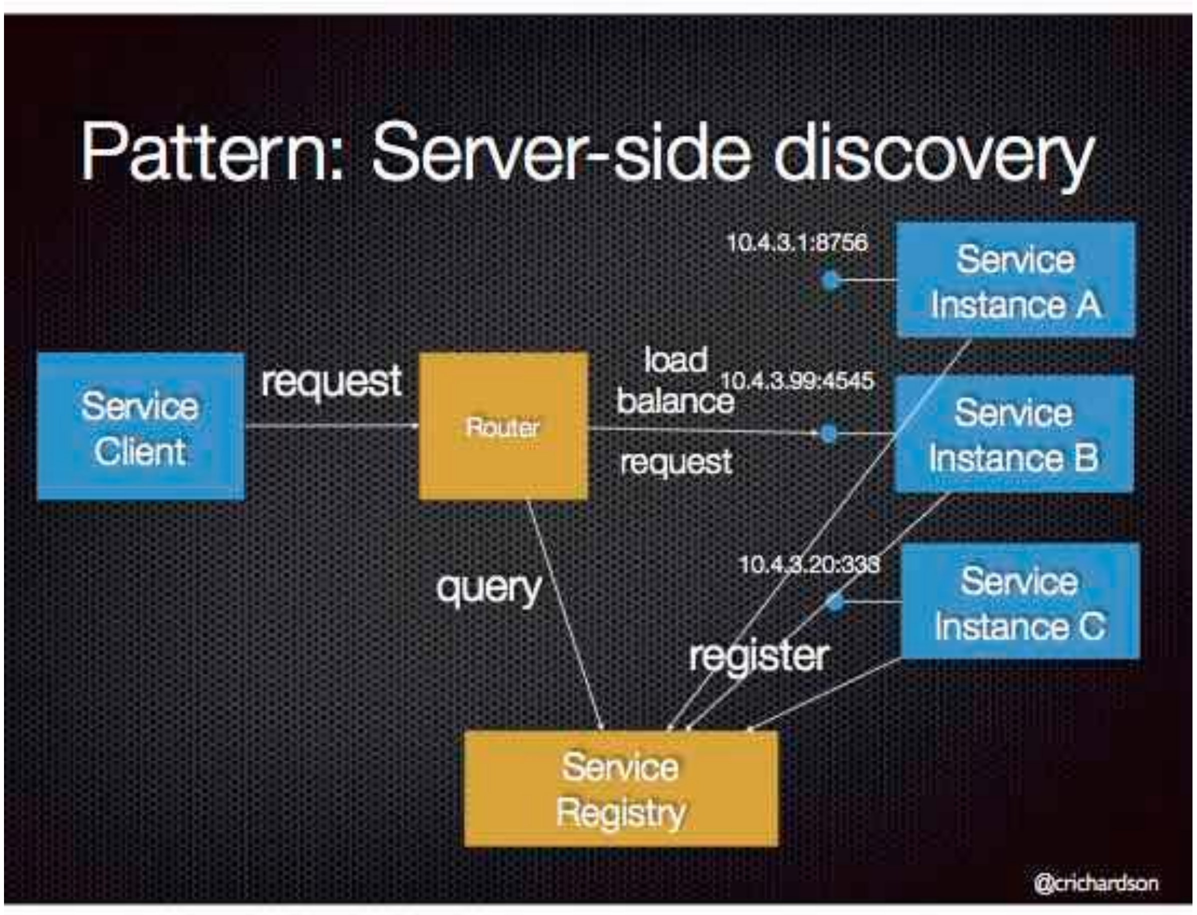


**Figure 15 - Server-side service discovery [34]**

There are 2 main ways that the service instances are registered with and deregistered from the service registry.

1. Self-registration pattern: is for service instances to register themselves with the service registry.
2. Third-party registration pattern: is for some other system component to handle the registration and deregistration on behalf of the service.

In this project, we use Apache Zookeeper as self-registration server-side service registry for Apache Kafka containers.

In some deployment environments, we need to set up our own service-discovery infrastructure using a service registry such as Netflix Eureka, etcd or Apache Zookeeper. In other deployment environments, service discovery is built in. For example, Kubernetes, Docker Swarm and Marathon handle service instance registration and deregistration.

Docker Swarm mode as of Docker 1.12, comes with Consul Service registration built-in to Docker engine. Docker Swarm services connected to the same Docker overlay network will be able to communicate with each other by service's name. All replicas of a service connect to the same overlay network(s). There is no explicit service registration required anymore and hence no need to host the service registry.

#### 4.2.2.6 DATABASE PER SERVICE

In a Microservices architecture, each microservice has its own **private datastore**. Different microservices might use different SQL and NoSQL databases. While this database architecture has significant benefits, it creates some distributed data management challenges.

1. how to implement business transactions that maintain consistency across multiple services
2. how to implement queries that retrieve data from multiple services.

For many applications, the solution is to use an event-driven architecture. One challenge with implementing an event-driven architecture is how to atomically update state and how to publish events. There are few ways to accomplish this, including using the database as a message queue, transaction log mining and event sourcing.

The simplicity of our design does not face any of the above 2 problems. There are 2 services that need their own private datastore:

1. User service – uses user-db service to store the user profile data. User service provides REST API interface for other services to consume the data in the user-db. No other service can directly access the user-db data.
2. Command Processor service – users command-db to store the message intent to command mapping. Again, this datastore is only for private consumption of command processor service alone. There is no other service that uses the data in the command db.

#### 4.2.2.7 DEPLOYING MICROSERVICES

Deploying a microservices application is challenging. There are tens or even hundreds of services written in a variety of languages and frameworks. Each one is a mini-application with its own specific deployment, resource, scaling and monitoring requirements. There are several microservice deployment patterns including:

1. Service instance per Virtual Machine, and
2. Service instance per Container
3. Serverless architecture like AWS Lambda. This is also known as Function as a Service (FaaS).

We chose to deploy all our services with option 2 (a service per container). Following are the benefits of this approach:

- It is straightforward to scale up and down a service by changing the number of container instances.
- The container encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in the same way.
- Each service instance is isolated
- A container imposes limits on the CPU and memory consumed by a service instance
- Containers are extremely fast to build and start. For example, it’s 100x faster to package an application as a Docker container than it is to package it as an AMI. Docker containers also start much faster than a VM since only the application process starts rather than an entire OS.

#### 4.2.2.8 LOG AGGREGATION

Use a centralized logging service that aggregates logs from each service instance. The users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs.

In this project, **Elasticsearch + Logstash + Kibana** (ELK stack) is used to aggregate the logs from all containers in the Docker swarm cluster.

#### 4.2.2.9 APPLICATION METRICS

Instrument a service to gather statistics about individual operations. Aggregate metrics in centralized metrics service, which provides reporting and alerting. There are two models for aggregating metrics:

- push - the service pushes metrics to the metrics service
- pull - the metrics services pull metrics from the service

In this project, **Prometheus** is used to collect application and node metrics. The node metrics are collected by using a **node-exporter** service. The container metrics are exported by container advisor (Google's **cAdvisor**) service. For both cases, push model is used where the metrics collectors push the metrics to Prometheus which stores the metrics. **Grafana** is used to then plot and build dashboards for monitoring application performance.

#### 4.2.2.10 AUDIT LOGGING

Record user activity in database. In this project, the user activity is logged in command DB. Each request and response are logged and can be queried for auditing.

#### 4.2.2.11 DISTRIBUTED TRACING

Instrument service code such that:

1. Assigns each external request a unique external request ID
2. Passes the external request ID to all services that are involved in handling the request
3. Includes the external request ID in all log messages

4. Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

This has the following benefits:

1. It provides useful insight into the behavior of the system including the sources of latency
2. It enables developers to see how an individual request is handled by searching across aggregated logs for its external request id.

In this project, request ID is generated at the Chatbot-Service and then transferred via the Kafka message payload to the command processor service. The request ID is persisted in the command DB and all logging corresponding to the servicing of that request logs the request ID with the time stamp so it is easy for the tracing the logs for a certain request.

#### 4.2.2.12 HEALTH-CHECK API

A service has a health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as

- the status of the connections to the infrastructure services used by the service instance
- the status of the host, e.g. disk space
- application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance

In this project, Dropwizard microservices framework provides a way to easily add health check implementation. In user-service for example, health check endpoint tests the connectivity of the service to the user-db.

### 4.2.3 End-to-End Control Flow

The below figure shows the sequence diagram of message that flows from Slack UI to AWS and the response from the backend back to the Slack user.

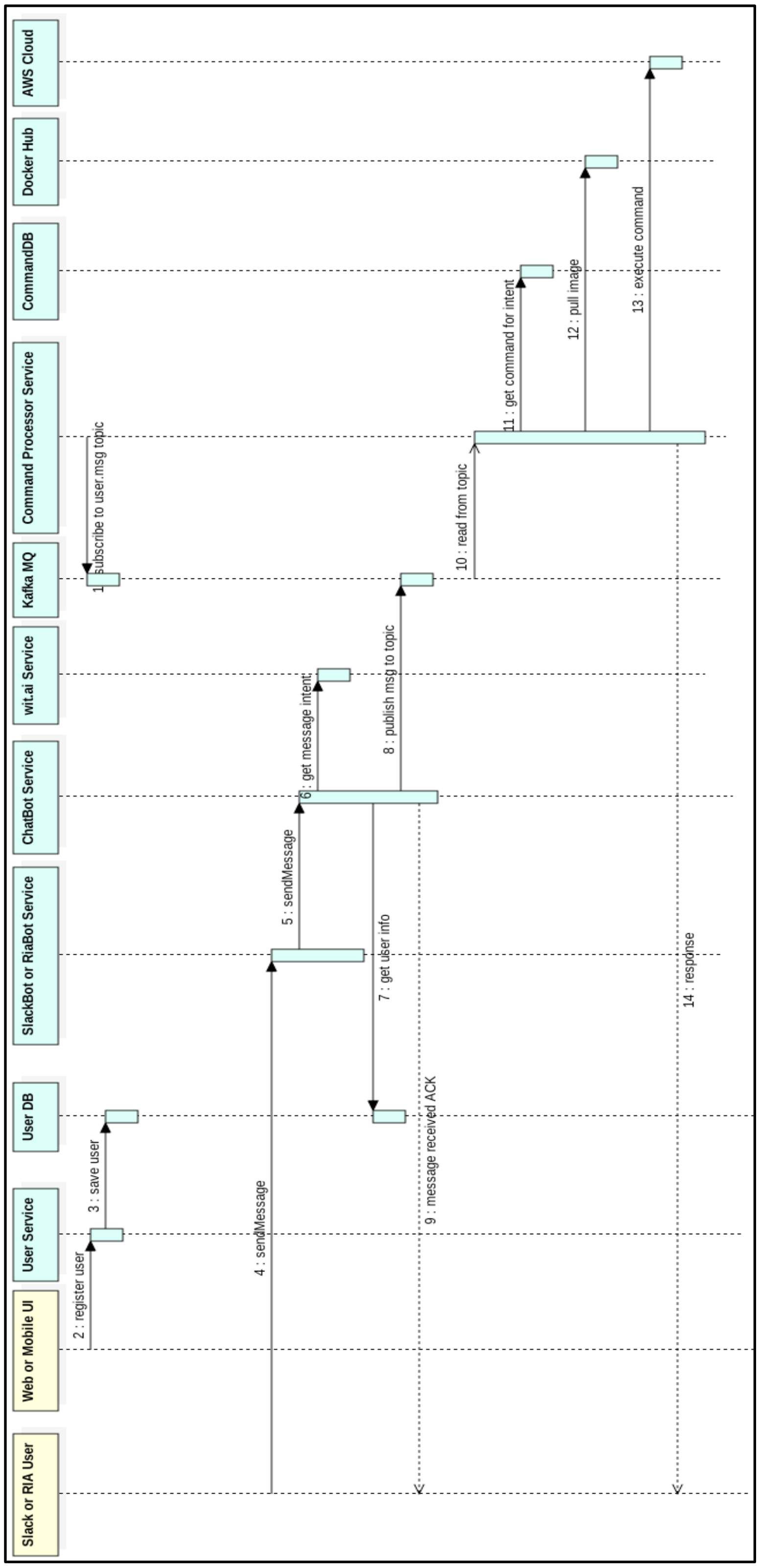


**Figure 16 - Chatbot Sequence Diagram**

Legends in the above sequence diagram are:

- Yellow colored entities are client side.
- Blue colored entities are server side.

Following are the steps shown in the above diagram:

1. Command processor service subscribes to the message queue topic (named “user.msg”).
2. User needs to register/sign up with the Amigo Chatbot service using the Web UI or Mobile App (or REST API client). Even though proxy service is not shown in the sequence diagram above, all requests are routed via the proxy service.
3. User provides a unique email ID identifying the user to the system, with slack user ID, RIA user ID and their AWS credentials. This info is persisted in user’s profile maintained in the user-db.
4. Slack or RIA (Raspberry Pi Intelligent Assistant) virtual assistant user sends a message to the slack or RIA bot service.
5. Slack or RIA bot service receives the message and forwards it to the Chatbot service.
6. Chatbot service gets the message intent from the wit.ai service.
7. If the message is valid then it is published on the user.msg topic on the Kafka message queue and a response that message is being processed is returned to the user with the generated request ID. If message is not valid (no intent found) then error is returned to the slack or RIA user.

8. Command processor service consumers read the message from the topic.
9. The message payload has the intent of the message using which Command DB is queried to find the command to execute for the intent.
10. The command is executed using the cloud provider’s CLI tool. The corresponding Docker image (which has the cloud provider’s CLI tool pre-installed) is pulled from the Docker hub registry.
11. The command is executed on the AWS cloud.
12. Response from the execution is returned to the slack user. In case of RIA user, the result needs to be polled actively with the request ID. Once the status of the response changes from IN_PROGRESS to one of SUCCESS or FAILURE, the result is communicated to the user.

### 4.2.4 User Service

User service is designed as a microservice that will provide user profile CRUD (Create, Read, Update and Delete) functionality.

A new user of Amigo Chatbot Service will be required to first register with the system using one of the 3 clients:

1. Web UI
2. Mobile App
3. REST API Client

User will be required to provide the following information:

1. User email
2. User name
3. User password
4. Slack user ID
5. RIA user ID
6. AWS Credentials
    a. AWS Region
    b. AWS Access Key ID
    c. AWS Secret Access Key

The above information will be persisted in the user DB.

User DB will run as a separate microservice container which is only accessed by the user service.

Any other service will access the user profile data via the REST APIs provided by user service.

Following REST APIs will be provided by the user service:

#### 4.2.4.1 CREATE USER

| | |
|---|---|
| **Protocol** | POST |
| **URI** | /api/v1.0/users |
| **Authentication** | Not Required |
| **Request Headers** | Content-Type: application/json |
| **Request body** | { <br> "email": "acme@sjsu.edu", <br> "name": "acme", |

| | |
|---|---|
| | "password":"pass", "slackUser":"acme", "riaId": "1", "awsCredentials": { "region": "us-west-2", "awsAccessKeyId": "secretKeyId", "awsSecretAccessKey": "secretAccessKey" } } |
| **Response status** | 201 Created |
| **Response body** | { "entity": "acme@sjsu.edu", "variant": { "language": null, "mediaType": { "type": "application", "subtype": "json", "parameters": {}, "wildcardType": false, "wildcardSubtype": false }, "encoding": null, "languageString": null }, "annotations": [], |

| | |
|---|---|
| | "language": null,<br>"encoding": null,<br>"mediaType": {<br>"type": "application",<br>"subtype": "json",<br>"parameters": {},<br>"wildcardType": false,<br>"wildcardSubtype": false<br>}<br>} |
| **Description** | Requests the user to be created. Server response is 201 (created) with entity ID that can be used by the client to retrieve the requested user. |

**4.2.4.2 GET USERS**

| | |
|---|---|
| **Protocol** | GET |
| **URI** | /api/v1.0/users |
| **Authentication** | Basic Authentication |
| **Request headers** | None |
| **Request body** | N/A |

| | |
|---|---|
| **Response status** | 200 OK |
| **Response body** | [<br>{<br>"email": "acme@sjsu.edu",<br>"name": "acme",<br>"password": "1a1dc91c907325c69271ddf0c944bc72",<br>"slackUser": "acme",<br>"riaId": "1",<br>"awsCredentials": {<br>"region": "us-west-2",<br>"awsAccessKeyId": "secretKeyId",<br>"awsSecretAccessKey": "secretAccessKey"<br>}<br>},<br>{<br>"email": "acme1@sjsu.edu",<br>"name": "acme1",<br>"password": "1a1dc91c907325c69271ddf0c944bc72",<br>"slackUser": "acme1",<br>"riaId": "1",<br>"awsCredentials": { |

| | |
|---|---|
| | "region": "us-west-2",<br><br>"awsAccessKeyId": "secretKeyId",<br><br>"awsSecretAccessKey": "secretAccessKey"<br><br>}<br><br>}<br><br>] |
| **Description** | Requests all users to be retrieved from the server. |

### 4.2.4.3 GET A USER BY EMAIL ID

| **Protocol** | GET |
|---|---|
| **URI** | /api/v1.0/users/{userId} |
| **Authentication** | Basic Authentication |
| **Request headers** | None |
| **Request body** | EMPTY |
| **Response status** | 200 OK |
| **Response body** | {<br><br>"email": "acme@sjsu.edu",<br><br>"name": "acme",<br><br>"password": "1a1dc91c907325c69271ddf0c944bc72", |

| | |
|---|---|
| | `"slackUser": "acme",`<br>`"riaId": "1",`<br>`"awsCredentials": {`<br>`"region": "us-west-2",`<br>`"awsAccessKeyId": "secretKeyId",`<br>`"awsSecretAccessKey": "secretAccessKey"`<br>`}`<br>`}` |
| **Description** | Requests to retrieve a user by email ID. |

### 4.2.4.4 UPDATE USER

| | |
|---|---|
| **Protocol** | PUT |
| **URI** | /api/v1.0/users/{userId} |
| **Authentication** | Basic Authentication |
| **Request body** | `{`<br>`"email": "acme@sjsu.edu",`<br>`"name": "acme",`<br>`"password":"pass",`<br>`"slackUser":"modified-acme",`<br>`"riaId": "1",`<br>`"awsCredentials": {`<br>`"region": "us-west-2",`<br>`"awsAccessKeyId": "secretKeyId",`<br>`"awsSecretAccessKey": "secretAccessKey"`<br>`}`<br>`}` |
| **Response status** | 200 |

| | |
|---|---|
| **Response body** | `{`<br>`  "email": "acme@sjsu.edu",`<br>`  "name": "acme",`<br>`  "password":"pass",`<br>`  "slackUser":"modified-acme",`<br>`  "riaId": "1",`<br>`  "awsCredentials": {`<br>`    "region": "us-west-2",`<br>`    "awsAccessKeyId": "secretKeyId",`<br>`    "awsSecretAccessKey": "secretAccessKey"`<br>`  }`<br>`}` |
| **Description** | Requests the user to be updated. Server response is 200 (ok). |

### 4.2.4.5 DELETE USER

| | |
|---|---|
| **Protocol** | DELETE |
| **URI** | /api/v1.0/users/{userId} |
| **Authentication** | Basic Authentication |
| **Request headers** | None |
| **Request body** | EMPTY |
| **Response status** | 200 |
| **Response body** | `{`<br>`  "entity": "acme@sjsu.edu",`<br>`  "variant": {`<br>`    "language": null,`<br>`    "mediaType": {` |

```
            "type": "application",
            "subtype": "json",
            "parameters": {},
            "wildcardType": false,
            "wildcardSubtype": false
        },
        "encoding": null,
        "languageString": null
    },
    "annotations": [],
    "language": null,
    "encoding": null,
    "mediaType": {
        "type": "application",
        "subtype": "json",
        "parameters": {},
        "wildcardType": false,
        "wildcardSubtype": false
    }
}
```

| | |
|---|---|
| **Description** | Requests the user to be removed. Server response is 200 (ok) meaning that the server performed the requested removal of user and response returns the ID of the deleted user. |

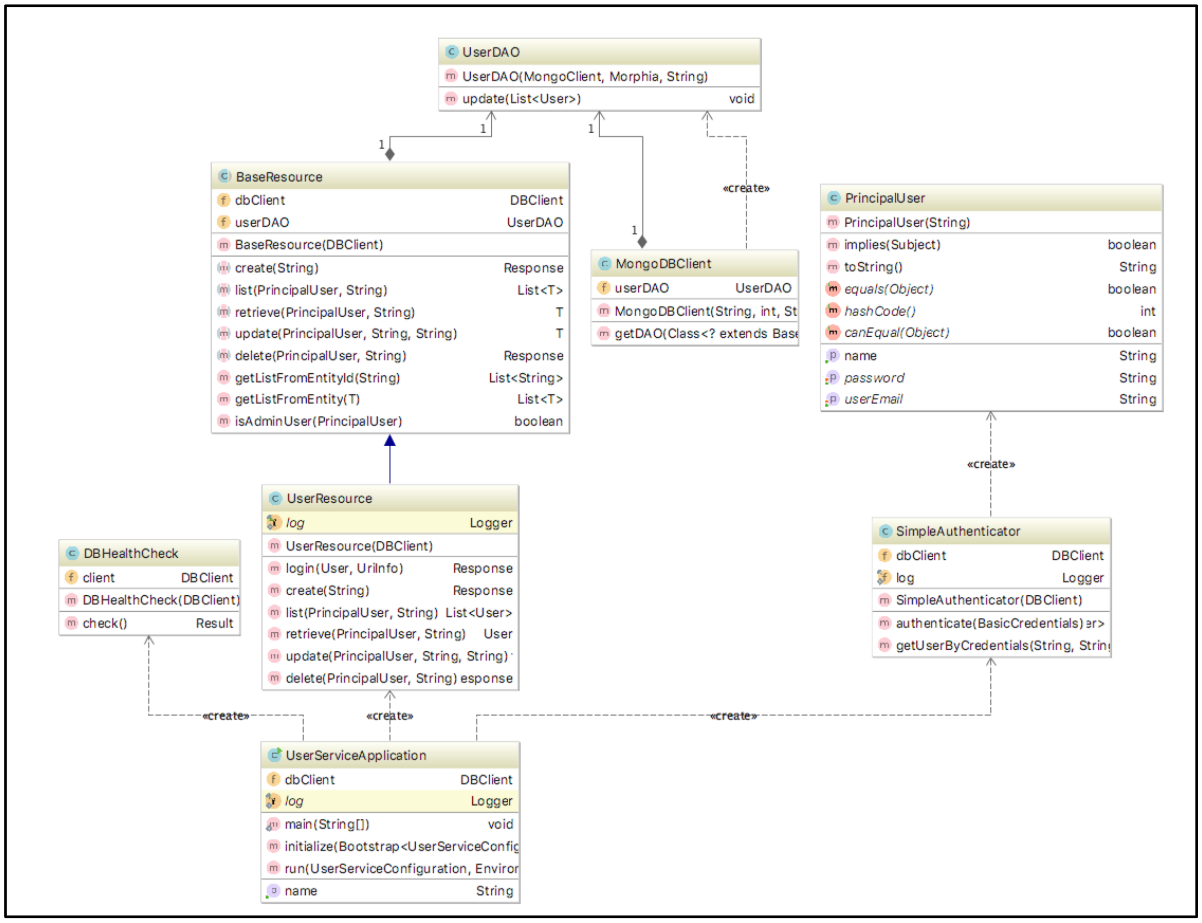

UserDAO
UserDAO(MongoClient, Morphia, String)
update(List<User>) void
BaseResource
dbClient DBClient
userDAO UserDAO
BaseResource(DBClient)
create(String) Response
list(PrincipalUser, String) List<T>
retrieve(PrincipalUser, String) T
update(PrincipalUser, String, String) T
delete(PrincipalUser, String) Response
getListFromEntityId(String) List<String>
getListFromEntity(T) List<T>
isAdminUser(PrincipalUser) boolean
MongoDBClient
userDAO UserDAO
MongoDBClient(String, int, St
getDAO(Class<? extends Base
PrincipalUser
PrincipalUser(String)
implies(Subject) boolean
toString() String
equals(Object) boolean
hashCode() int
canEqual(Object) boolean
name String
password String
userEmail String
UserResource
log Logger
UserResource(DBClient)
login(User, UriInfo) Response
create(String) Response
list(PrincipalUser, String) List<User>
retrieve(PrincipalUser, String) User
update(PrincipalUser, String, String)
delete(PrincipalUser, String) esponse
DBHealthCheck
client DBClient
DBHealthCheck(DBClient)
check() Result
SimpleAuthenticator
dbClient DBClient
log Logger
SimpleAuthenticator(DBClient)
authenticate(BasicCredentials) er>
getUserByCredentials(String, Strin
UserServiceApplication
dbClient DBClient
log Logger
main(String[]) void
initialize(Bootstrap<UserServiceConfig
run(UserServiceConfiguration, Enviror
name String
«create»
«create»
«create»
«create»
«create»
1
1
1
1

**Figure 17 - User Service Class Diagram**

Above class diagram shows the following classes:

| **Class** | **Description** |
|---|---|

| | |
|---|---|
| **BaseResource** | It’s an abstract base class which defines common CRUD methods to be implemented by any REST resource. |
| **DBHealthCheck** | Implements health check by checking the connectivity to user DB. |
| **MongoDBClient** | Implements the DB client for user DB. |
| **PrincipalUser** | Represents an authenticated user. |
| **SimpleAuthentication** | Implements basic authentication. |
| **UserDAO** | User DB Data access object. |
| **UserResource** | Implements the REST endpoints for the user service. |
| **UserServiceApplication** | Main entry point to the service. It initializes the service by loading the externalized configuration in YAML configuration file, creates a DB client, initializes the REST endpoints, starts up the Jetty embedded web server. |

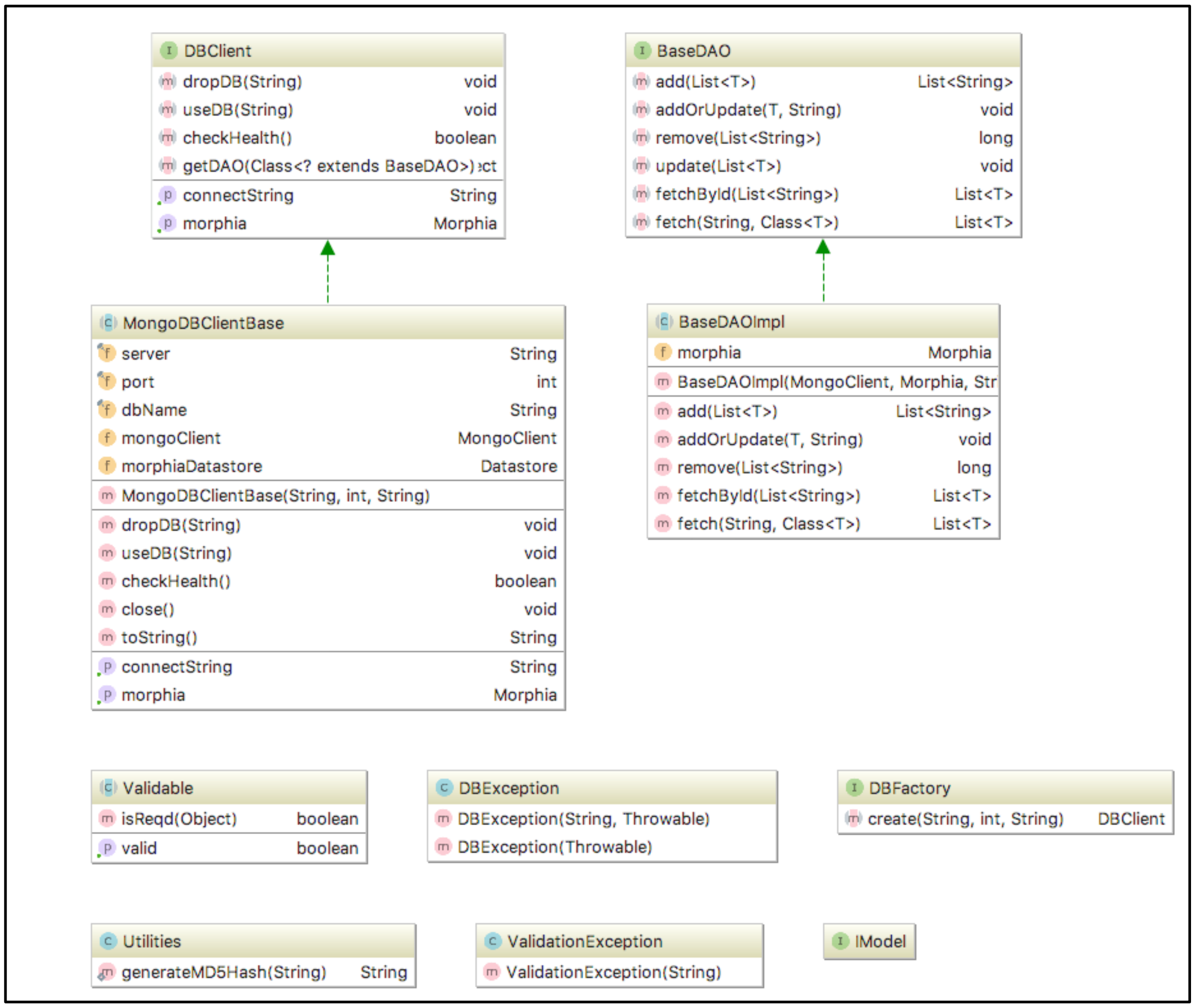

DBClient
dropDB(String) void
useDB(String) void
checkHealth() boolean
getDAO(Class<? extends BaseDAO>) ›ct
connectString String
morphia Morphia
BaseDAO
add(List<T>) List<String>
addOrUpdate(T, String) void
remove(List<String>) long
update(List<T>) void
fetchById(List<String>) List<T>
fetch(String, Class<T>) List<T>
MongoDBClientBase
server String
port int
dbName String
mongoClient MongoClient
morphiaDatastore Datastore
MongoDBClientBase(String, int, String)
dropDB(String) void
useDB(String) void
checkHealth() boolean
close() void
toString() String
connectString String
morphia Morphia
BaseDAOImpl
morphia Morphia
BaseDAOImpl(MongoClient, Morphia, Str
add(List<T>) List<String>
addOrUpdate(T, String) void
remove(List<String>) long
fetchById(List<String>) List<T>
fetch(String, Class<T>) List<T>
Validable
isReqd(Object) boolean
valid boolean
DBException
DBException(String, Throwable)
DBException(Throwable)
DBFactory
create(String, int, String) DBClient
Utilities
generateMD5Hash(String) String
ValidationException
ValidationException(String)
IModel

**Figure 18 - Common DB Module Class Diagram**

Above class diagram shows the common DB module classes that will be used as a shared library by services that need to access DB services. This is a common module for DB access that will be used by both user service and command processor service.

Following are the classes shown in the above diagram:

| **Class** | **Description** |
|---|---|
| **BaseDAO** | Common contract for all DAOs. |
| **BaseDAOImpl** | Generic DAO implementation for all mongoDB entities. It implements all methods except update method that needs to be implemented for each entity in its DAO class. |
| **DBClient** | This class encapsulates DB client operations that are not entity specific. |
| **DBFactory** | A factory class to create a new DB client. Each DB Client represents a certain DB type. So, with factory it is easy to replace actual DB with a mock DB or a different type of DB. |
| **IModel** | A marker interface for domain models. |
| **MongoDBClientBase** | Common base class for MongoDB Client. |
| **Utilities** | This class has utility methods like generateMD5Hash for encrypting passwords or any text before saving them in DB. |
| **Validable** | An abstract class that is inherited from by model classes that are validated if they have all required fields as not null etc. |
| **ValidationException** | If there is a validation error this exception is thrown. |

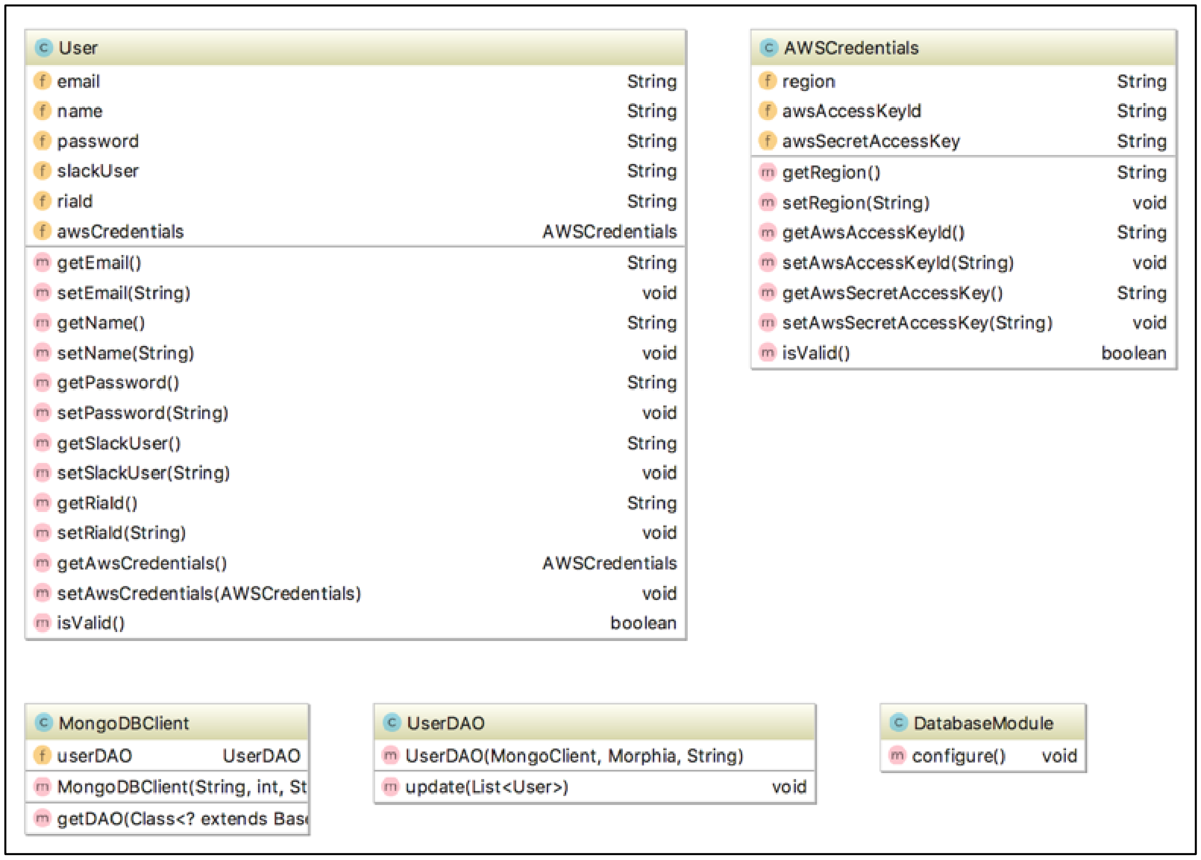


**Figure 19 - Class Diagram for User DB**

The above class diagrams are for classes that are specific to user DB:

| **Class** | **Description** |
|---|---|
| **AWSCredentials** | Models the AWS credentials provided by the user during registration. This is an embedded entity within User entity. |
| **DatabaseModule** | This class is a Google Guice IoC (Inversion of Control) container's module that maps the MongoDBClient class as implementation of DBClient type. |

| | |
|---|---|
| **MongoDBClient** | This class provides a handle to all DAO classes of all user DB entities. |
| **User** | Models the user data provided by the user during registration. |
| **UserDAO** | Represents DAO for User entity. |

### 4.2.5 Command Processor Service

Shown below is the class diagram for the command processor service.

It shows the design is extensible to support multiple cloud providers. The Docker Task loads the Docker image from Docker hub registry based on the information read from the Command DB. So, the DockerTask is agnostic to any cloud provider or the type of command it is executing. Till the time the intent has a mapping to a Docker image and the command string to be passed as an argument with the Docker image while running the Docker container, it can execute any command against any cloud provider or even an entirely different task than cloud operations management.

This design of using Docker image to bundle the existing AWSCLI client tool has an additional benefit that we can execute any command on behalf of the slack user that is supported by the awscli command. This same idea can be extended to managing other cloud provider services from a chatbot interface and with minimal code change a new provider can be supported on which we can perform all operations that its CLI client tool can perform.

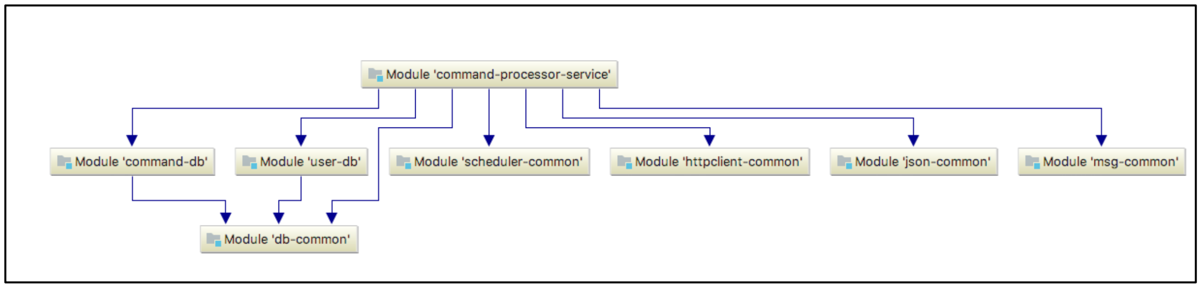


**Figure 20 - Command Processor Service module and its dependencies**

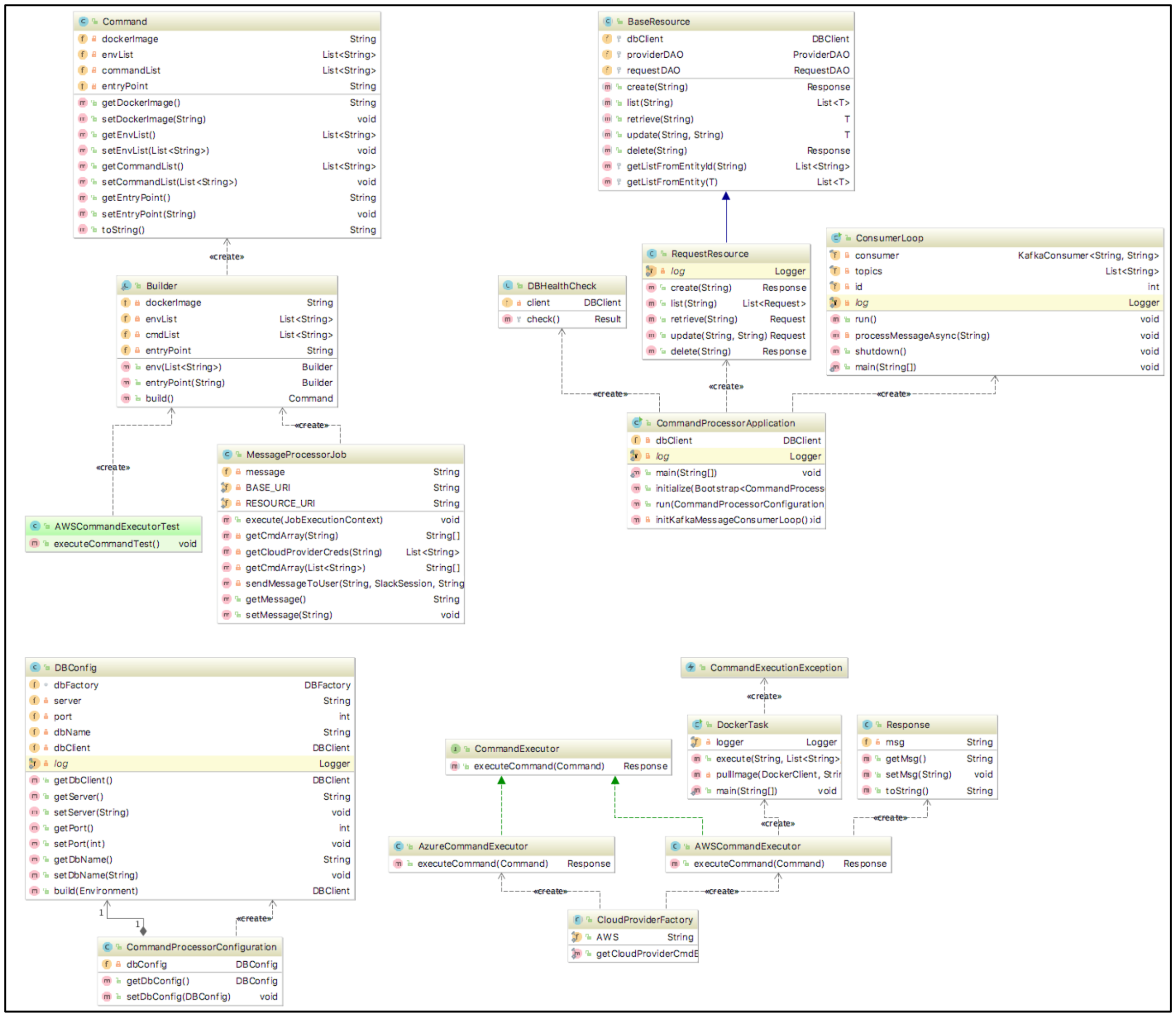


**Figure 21 - Command Processor Class Diagram**

### 4.2.5.1 GET RESULT

| | |
|---|---|
| **Protocol** | GET |
| **URI** | /api/v1.0/cmd/response/{requestId} |
| **Authentication** | Basic Authentication |
| **Request Headers** | Content-Type: application/json |
| **Request body** | None |
| **Response status** | 200 OK |
| **Response body** | { "_id" : "84a5-bc63ad413f0a", "startTime" : ISODate("2017-04-17T02:09:28.978Z"), "respRecvdTime" : ISODate("2017-04-17T02:09:28.978Z"), "commandExecuted" : "aws iam list-users", "resp" : "{\n \"Users\": [\n {\n \"UserName\": \"admin\",\n \"PasswordLastUsed\": \"2016-02-18T04:21:55Z\",\n \"CreateDate\": \"2016-02-13T00:20:57Z\",\n \"UserId\": \"AIDIDID\",\n \"Path\": \"/\",\n \"Arn\": \"arn:aws:iam::064674:user/admin\"\n },\n {\n \"UserName\": \"rwatsh\",\n \"Path\": \"/\",\n \"CreateDate\": \"2016-09-17T16:08:47Z\",\n |

| | |
|---|---|
| | \"UserId\": \"DGDGFG\",\n \"Arn\": \"arn:aws:iam::064674:user/rwatsh\"\n }\n ]\n}", "status" : "SUCCESS" } |
| **Description** | Requests to get the result of execution of the command. |

### 4.2.6 Slack-Bot Service

The slack bot service is responsible for listening to all incoming messages from slack messenger either sent to the channel where amigo chatbot is a participant or sent as a direct message to the chatbot. It then parses the message that have been meant for amigo chatbot explicitly by looking for messages that start with @amigo. It then gets the rest of the text and invokes the wit.ai service to parse the message and get its intent. If wit.ai service could not get the intent from the message, then error is returned to the user. If the service could parse the message and get its intent, then it returns the intent to the service. In such a case a new message payload is created and published to Kafka message queue topic named “user.msg”.

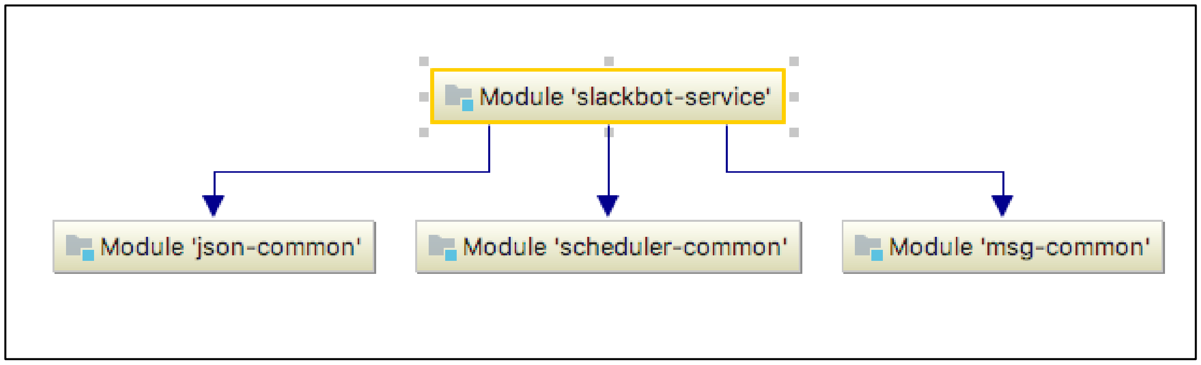


**Figure 22 - Slackbot service module dependencies**

Below class diagram shows some of the essential classes of slack bot service.

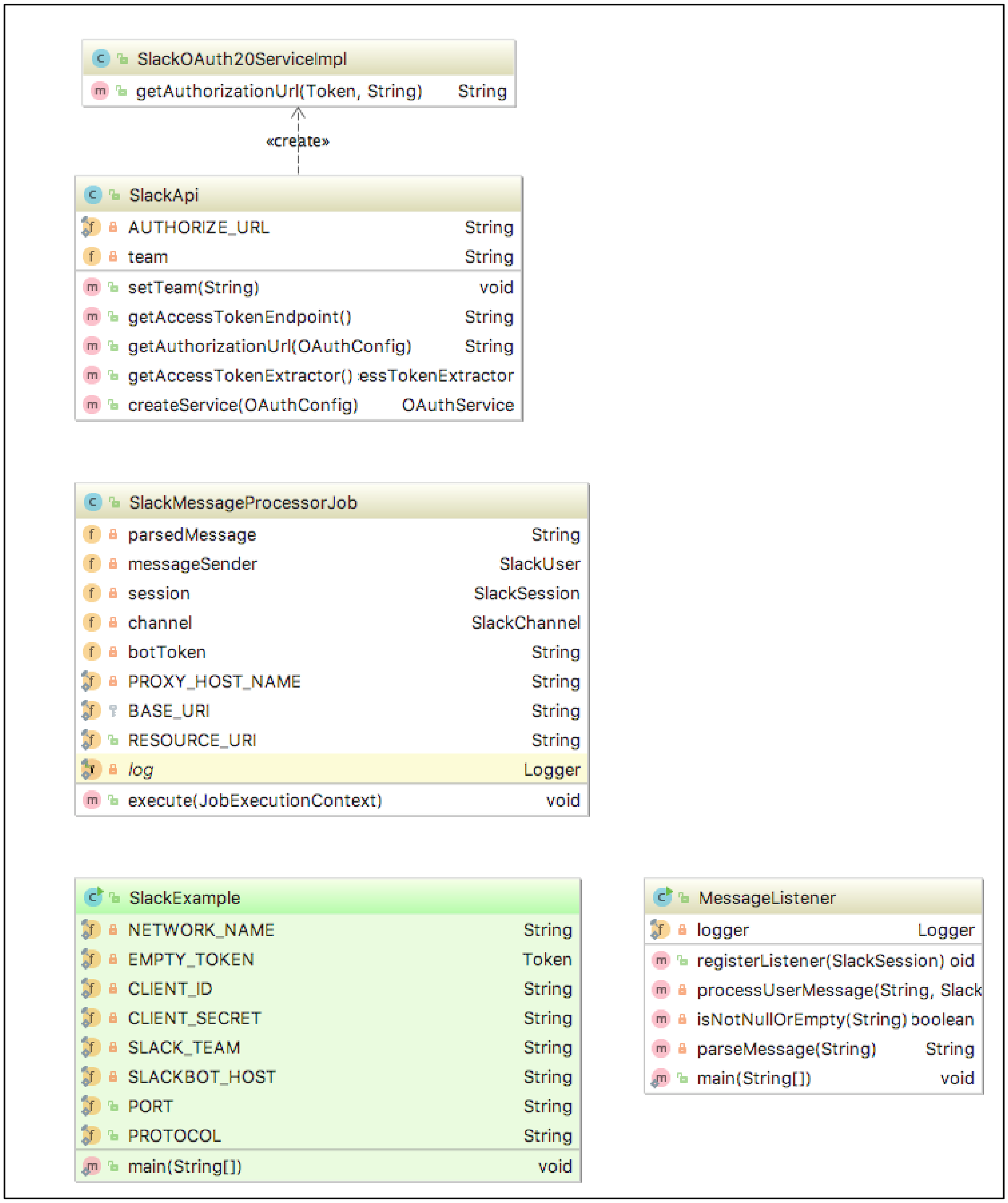


**Figure 23 - Slack Bot Class Diagram**

### 4.2.7 Chat-Bot Service

Chat-bot service receives the message from the various adapters for different client types. The received message is then sent to the wit.ai service. At the wit.ai service the message is parsed and intent of the message is inferred. If the intent could not be inferred then wit.ai returns the entire message as-is. The inferred intent is then published to the **user.msg** Kafka topic.

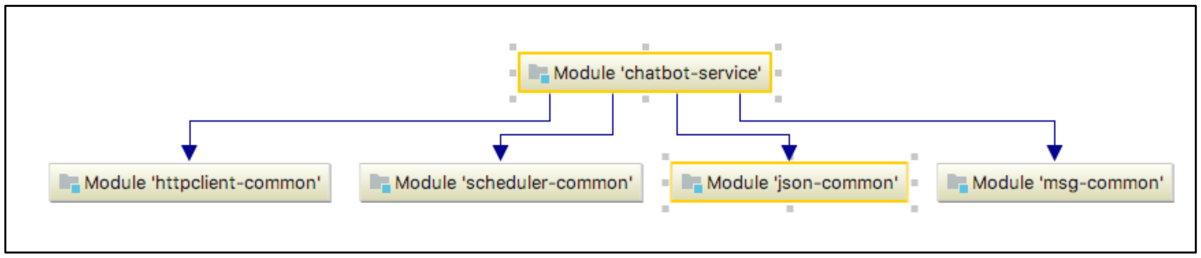


**Figure 24 - Chatbot service module dependencies**

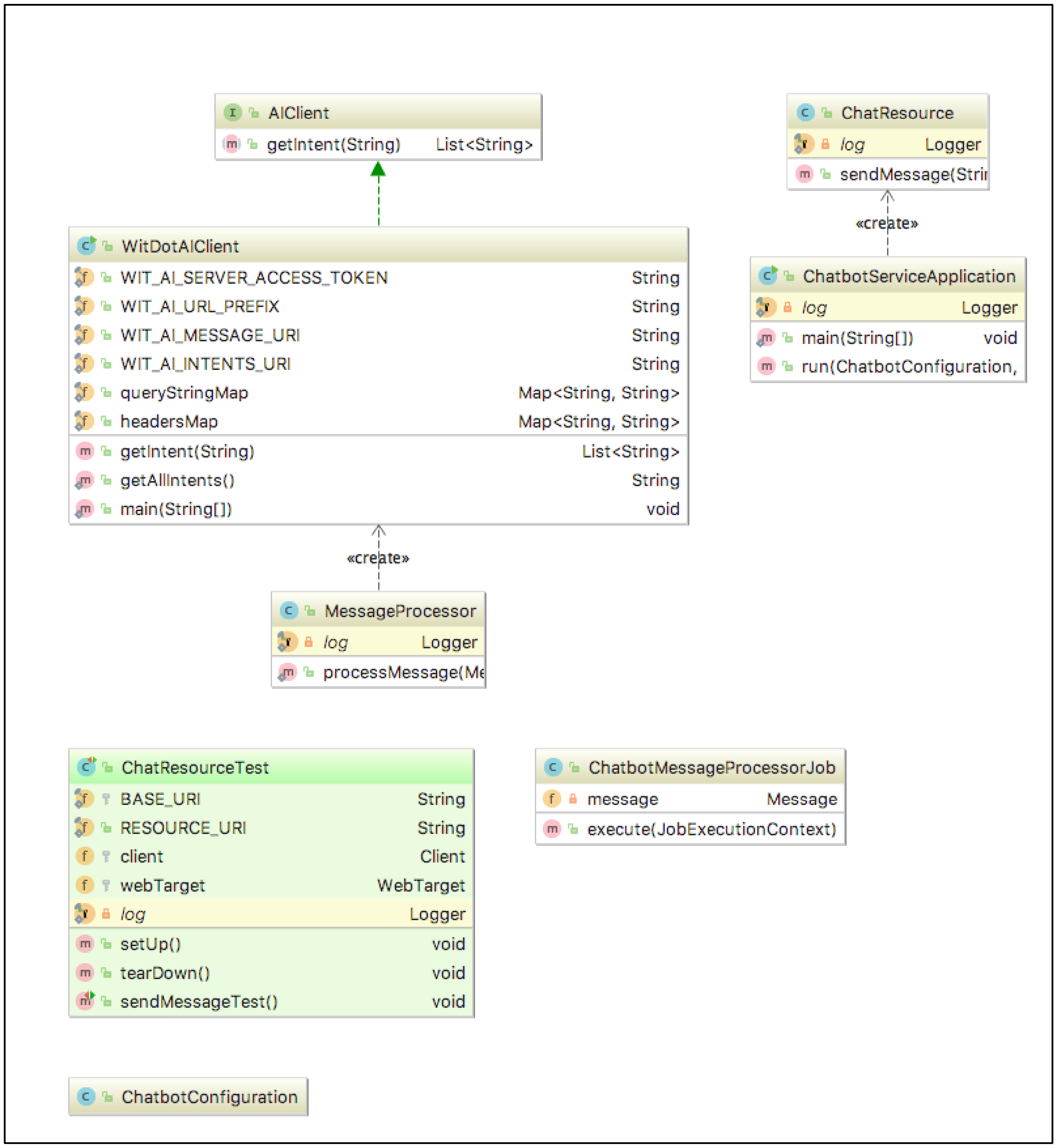


**Figure 25 – Chat-Bot Service Class Diagram**

### 4.2.7.1 SEND NEW MESSAGE

| | |
|---|---|
| **Protocol** | POST |

| | |
|---|---|
| **URI** | /api/v1.0/chat |
| **Authentication** | Basic Authentication |
| **Request Headers** | Content-Type: application/json |
| **Request body** | { <br> "botType" : "SLACK", <br> "msgReceivedTime" : "Mon May 08 00:43:38 PDT 2017", <br> "userEmail" : "watsh.rajneesh@sjsu.edu", <br> "userName" : "rwatsh", <br> "content" : " aws help", <br> "intent" : [], <br> "requestId" : "", <br> "channelId" : "D558L6680", <br> "slackBotToken" : "xoxb-175292210080-qiwVClAzMfwVT" <br> } <br><br> PS: Intent and requestID are filled in at Chatbot service. |
| **Response status** | 202 Accepted for success <br><br> 400 Bad Request for failure (unable to process message) |
| **Response body** | None – for success <br><br> Error message – for failure |
| **Description** | Requests the message to be processed. Server response is 202 (accepted) with request ID that can be used by the client to retrieve the response. |

### 4.2.8 RIA-BOT SERVICE

The Raspberry Pi Intelligent Assistant (RIA) is a voice controlled interface to the chatbot service. The RIA bot service is an adapter for receiving messages from RIA devices. It then forwards those to the generic Chatbot service where the intent of the message is inferred and message published on the message queue for further processing by command processor service asynchronously.

Following are the module dependencies for the RIA bot service module:

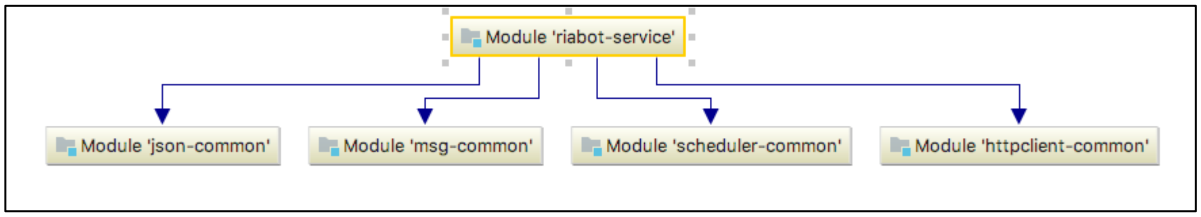


**Figure 26 - RiaBot Service module dependencies**

Following is class diagram for RIA bot service:

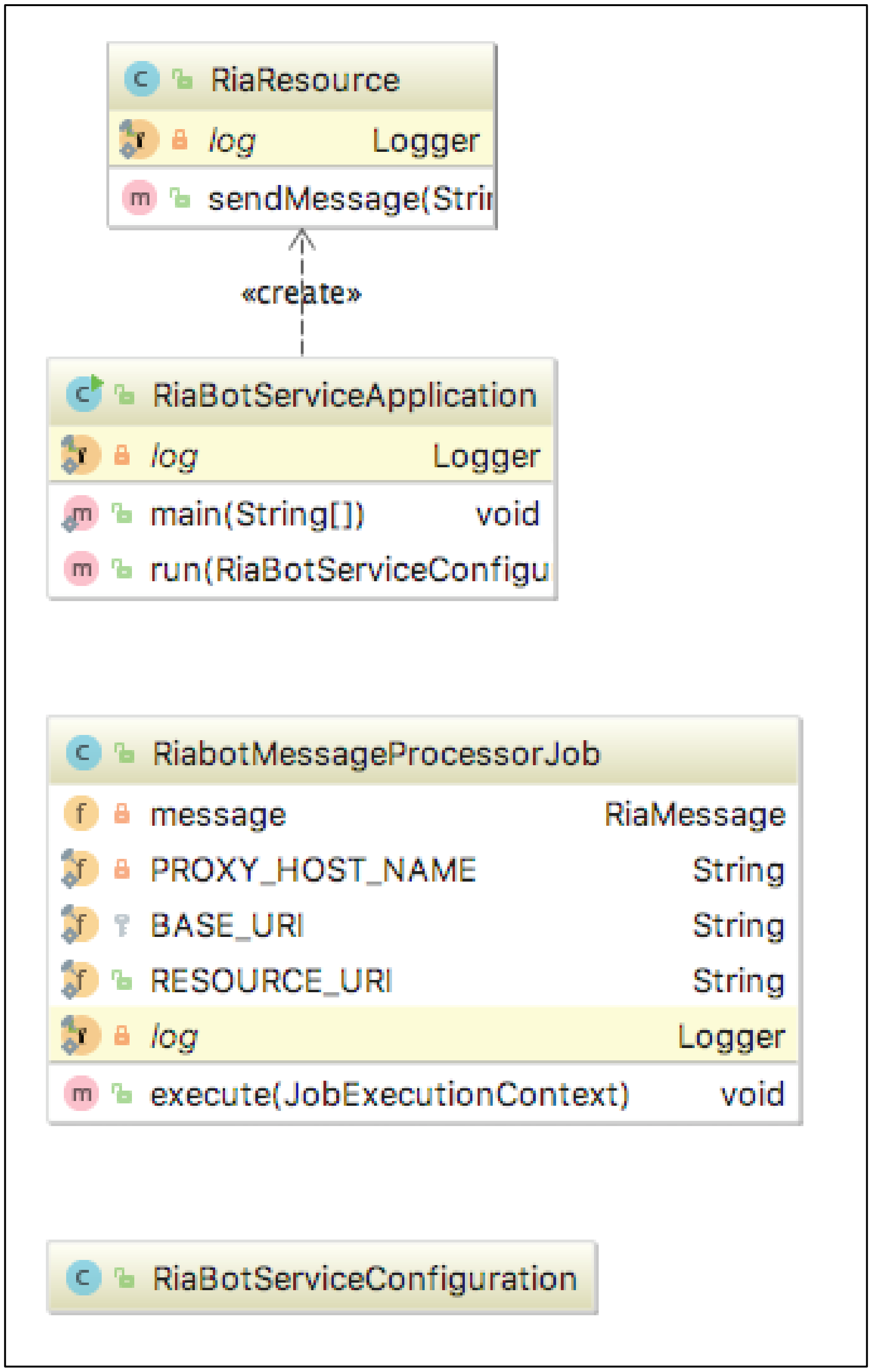


**Figure 27 - RiaBot Service class diagram**

Following is the REST endpoint exposed by RIA bot service for the RIA devices to send the messages to:

| | |
|---|---|
| **Protocol** | POST |
| **URI** | /api/v1.0/ria |
| **Authentication** | Basic Authentication |
| **Request Headers** | Content-Type: application/json |
| **Request body** | `{ "content" : "aws help", "riaId" : "001", }`<br><br>PS: RIA ID identifies the device with a unique ID. This ID should be specified as part of the user's profile during registration. |
| **Response status** | 202 Accepted for success<br><br>400 Bad Request for failure (unable to process message) |
| **Response body** | None – for success<br><br>Error message – for failure |
| **Description** | Requests the message to be processed. Server response is 202 (accepted) with request ID that can be used by the client to retrieve the response. |

## 4.3 DATA-TIER DESIGN

This section describes the DB schema design. There are 2 DBs in the system and each one is private to its respective service. Other services should not access that DB's data directly but only do so via the REST APIs of the service consuming that DB.

### 4.3.1 USER-DB

This DB is consumed by user service and stores the user's profile data. User service provides the CRUD interface to the DB's data via its REST APIs that is described in above section.

This DB is a NoSQL mongoDB and there is just one collection **users** and following is an example document in that collection which shows the information that is persisted in the DB per user.

#### 4.3.1.1 USERS COLLECTION

```
> db.users.find().pretty()

{

        "_id" : "test@gmail.com",

        "name" : "testUser",

        "password" : "password",
        "slackUser" : "watsh.rajneesh@sjsu.edu",

        "riaId" : "101",

        "awsCredentials" : {

                "region" : "us-west-2",

                "awsAccessKeyId" : "abc",

                "awsSecretAccessKey" : "def"

        }

}
```

### 4.3.2 COMMAND-DB

This DB is consumed by command processor service and it stores the mapping between intent of the message for a given cloud provider and the command that needs to be executed on it.

#### 4.3.2.1 PROVIDERS COLLECTION

Providers collection contains provider name as the key, a docker image corresponding to the provider which consists of the provider specific CLI tool pre-installed in the image and commands list which is a mapping of intent to command.

If no mapping is found for the intent then the incoming message is passed as command to the CLI.

```
> db.providers.find().pretty()
{
        "_id" : "aws",
        "dockerImage" : "sjsucohort6/docker_awscli",
        "commands" : [
                {
                        "intent" : "iam list users",
                        "cmdList" : [
                                "iam",
                                "list-users"
                        ]
                }
        ]
}
```

### 4.3.2.2 REQUESTS COLLECTION

The requests collection consists of request ID, start time of request, command to be executed, time at which the response is received, the status of the command execution and the command result.

```
> db.requests.find().pretty()
```

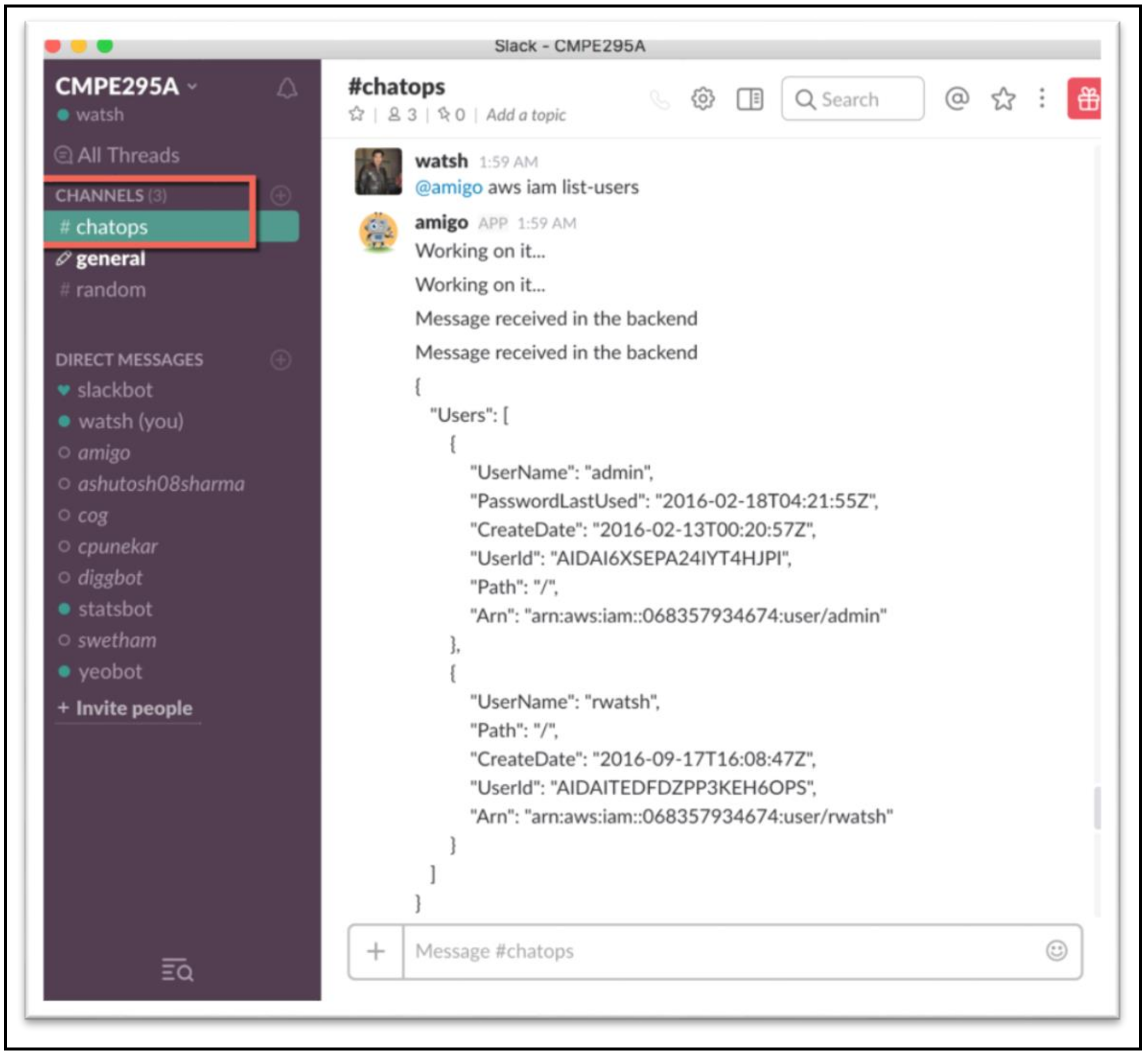
```
{
	"_id" : "84a5-bc63ad413f0a",
	"startTime" : ISODate("2017-04-17T02:09:28.978Z"),
	"respRecvdTime" : ISODate("2017-04-17T02:09:28.978Z"),
	"commandExecuted" : "aws iam list-users",
	"resp" : "{\n    \"Users\": [\n        {\n            \"UserName\": \"admin\",\n \"PasswordLastUsed\": \"2016-02-18T04:21:55Z\",\n          \"CreateDate\": \"2016-02-13T00:20:57Z\",\n         \"UserId\": \"AIDIDID\",\n         \"Path\": \"/\",\n         \"Arn\": \"arn:aws:iam::064674:user/admin\"\n      },\n      {\n         \"UserName\": \"rwatsh\",\n \"Path\": \"/\",\n          \"CreateDate\": \"2016-09-17T16:08:47Z\",\n          \"UserId\": \"DGDGFG\",\n          \"Arn\": \"arn:aws:iam::064674:user/rwatsh\"\n      }\n   ]\n}",
	"status" : "SUCCESS"
}
```

# Chapter 5. PROJECT IMPLEMENTATION

## 5.1 CLIENT IMPLEMENTATION

### 5.1.1 SLACK MESSENGER INTEGRATION

The figure below show user chatting with the bot in Slack Messenger in a channel named chatops. A channel can have multiple users conversing with the bot at the same time.



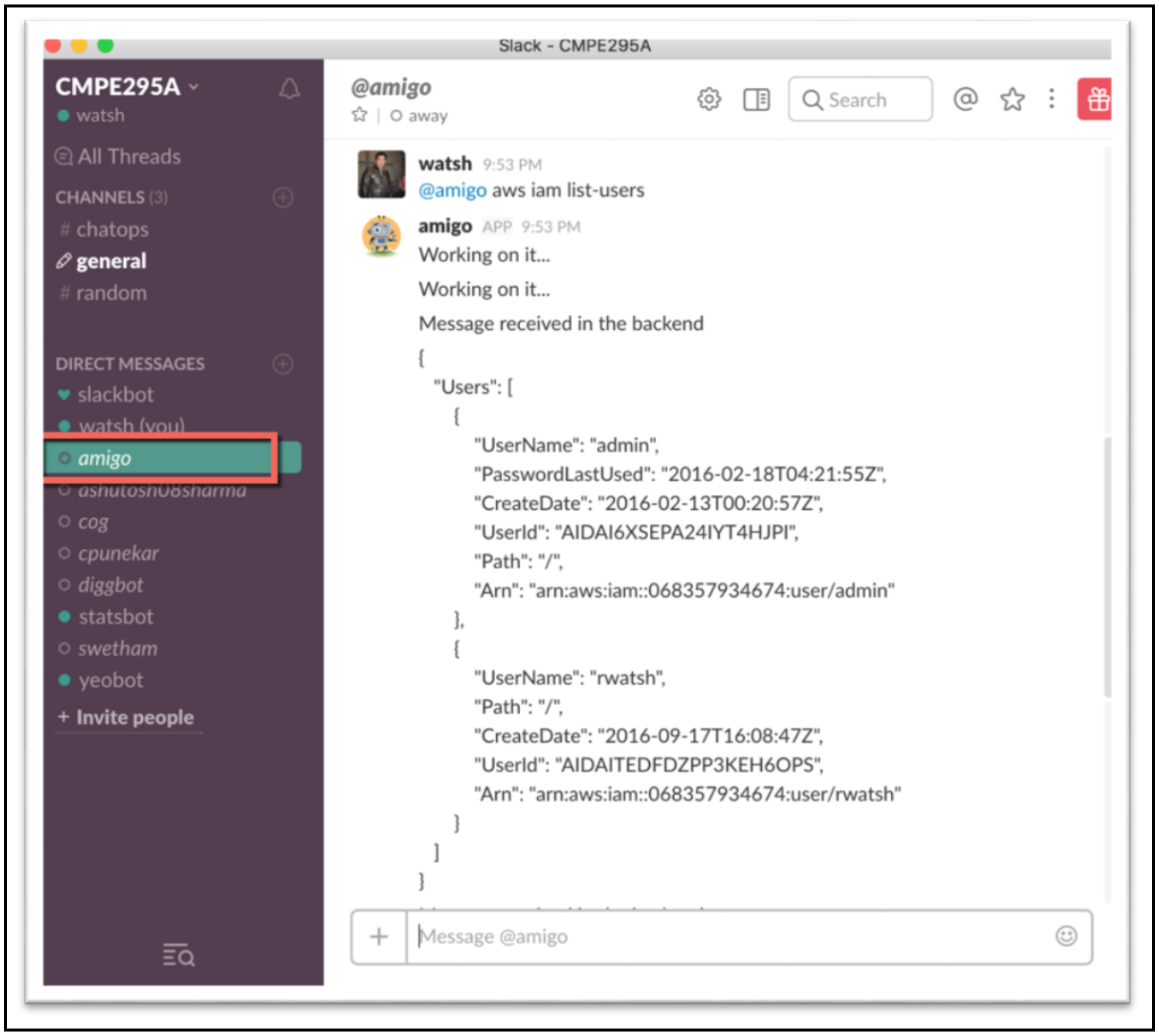
watsh 1:59 AM
@amigo aws iam list-users

amigo APP 1:59 AM
Working on it...
Working on it...
Message received in the backend
Message received in the backend

```
{
  "Users": [
    {
      "UserName": "admin",
      "PasswordLastUsed": "2016-02-18T04:21:55Z",
      "CreateDate": "2016-02-13T00:20:57Z",
      "UserId": "AIDAI6XSEPA24IYT4HJPI",
      "Path": "/",
      "Arn": "arn:aws:iam::068357934674:user/admin"
    },
    {
      "UserName": "rwatsh",
      "Path": "/",
      "CreateDate": "2016-09-17T16:08:47Z",
      "UserId": "AIDAITEDFDZPP3KEH6OPS",
      "Arn": "arn:aws:iam::068357934674:user/rwatsh"
    }
  ]
}
```

**Figure 28 - Conversation with Chatbot in a Channel named "Chatops"**

The figure below show the user conversing with the amigo chatbot in direct message mode. Direct message mode is when user and the chatbot are the exchanging messages without any other users participating in the conversation or watching the messages exchanged.



watsh 9:53 PM
@amigo aws iam list-users

amigo APP 9:53 PM
Working on it...
Working on it...
Message received in the backend

```
{
  "Users": [
    {
      "UserName": "admin",
      "PasswordLastUsed": "2016-02-18T04:21:55Z",
      "CreateDate": "2016-02-13T00:20:57Z",
      "UserId": "AIDAI6XSEPA24IYT4HJPI",
      "Path": "/",
      "Arn": "arn:aws:iam::068357934674:user/admin"
    },
    {
      "UserName": "rwatsh",
      "Path": "/",
      "CreateDate": "2016-09-17T16:08:47Z",
      "UserId": "AIDAITEDFDZPP3KEH6OPS",
      "Arn": "arn:aws:iam::068357934674:user/rwatsh"
    }
  ]
}
```

**Figure 29 - Direct message conversation with Chatbot**

## 5.1.2 Web UI

Web User Interface is the first step for a user to start using Amigo chatbot services. A user should sign up on the Web UI and provide all the details that can be utilized by us to make them easily access Amigo chatbot.

Following are the screens of the Web UI:

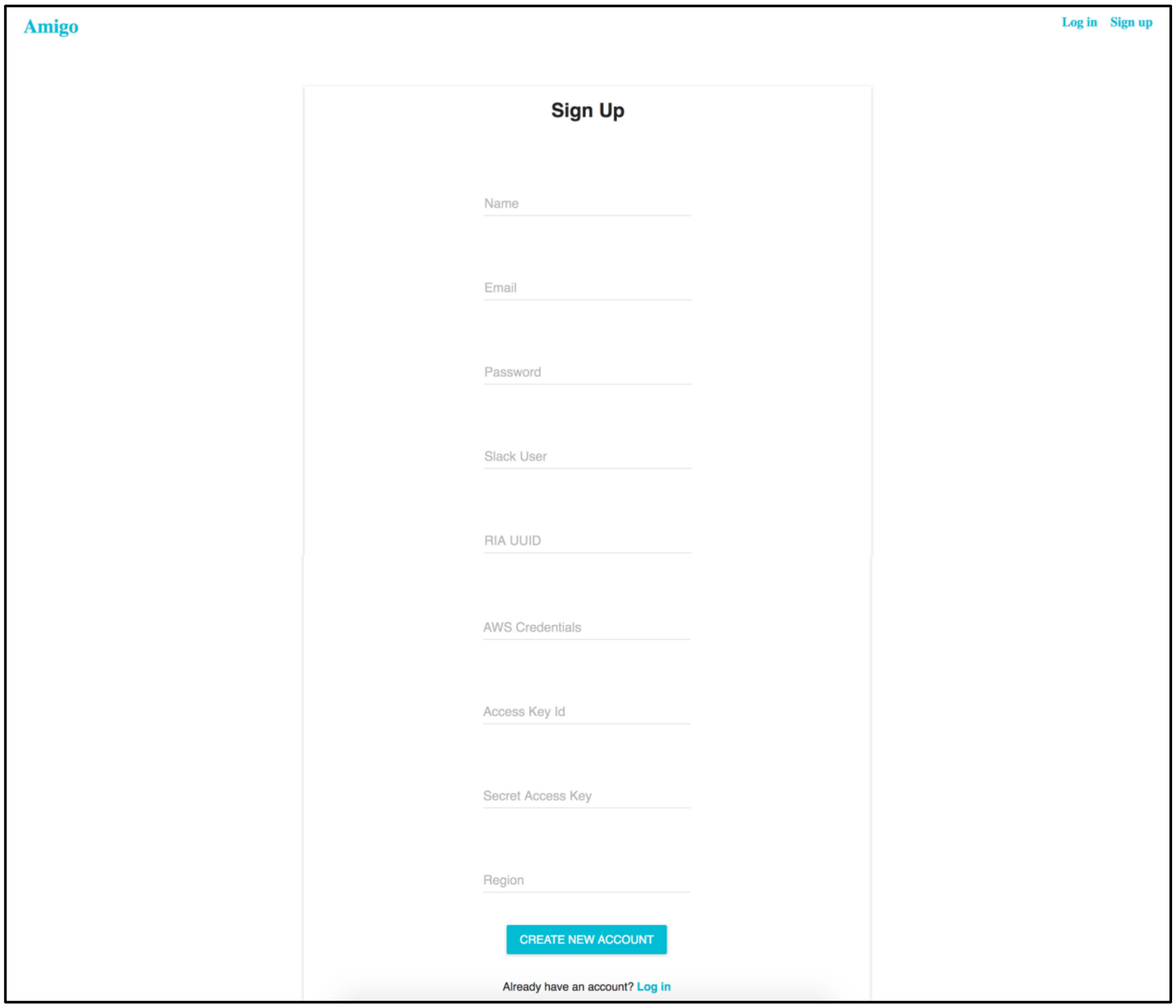


**Figure 30 - Sign Up Screen**

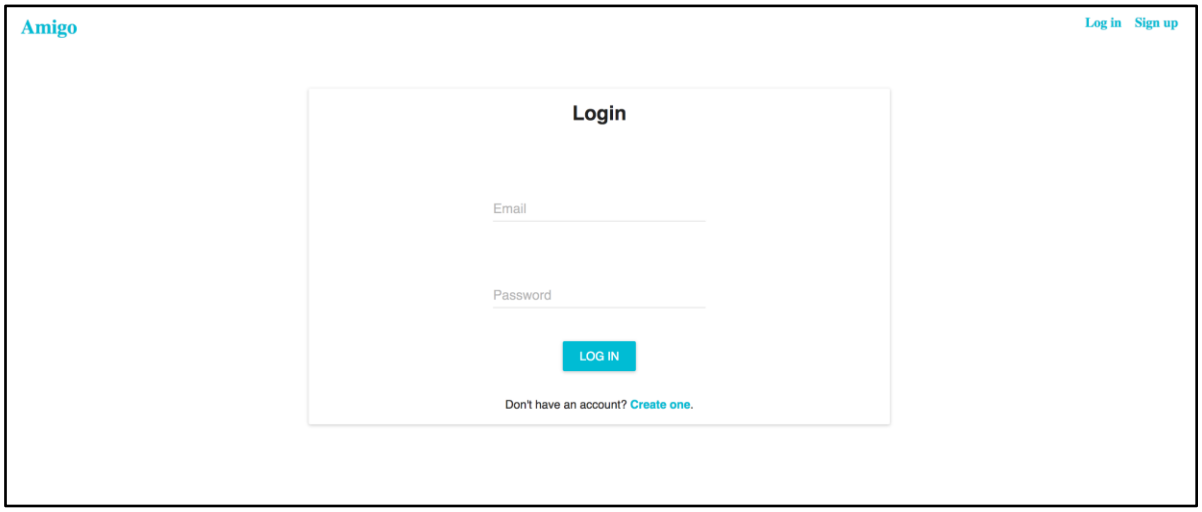
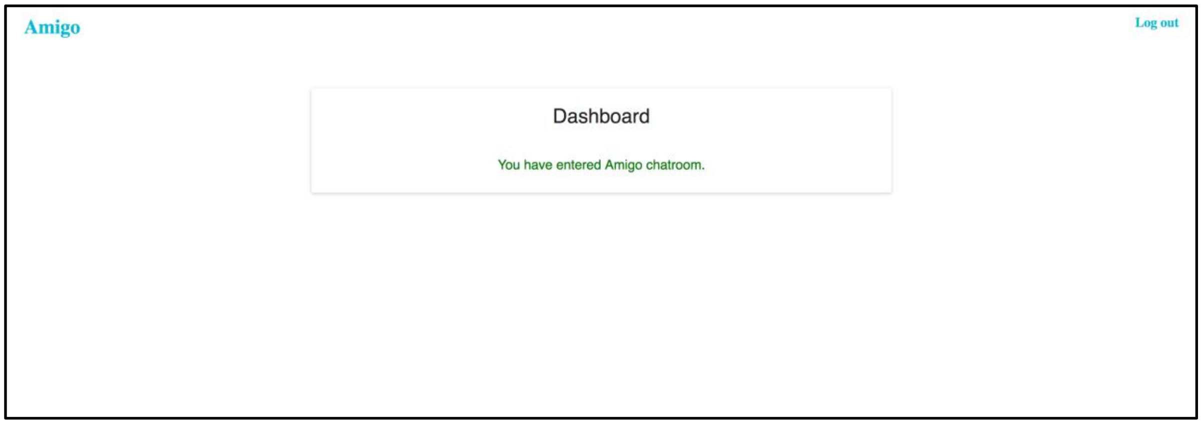


**Figure 31 – Login Screen**



You have entered Amigo chatroom.

**Figure 32 – Chat Room Screen**

## 5.1.3 Mobile Application

To access Amigo service, user must register to our services through web portal or mobile app. For mobile app, we are using react native so it is easy to create app for both iOS and Android platforms.

Below are some implementations from applications:

‘’

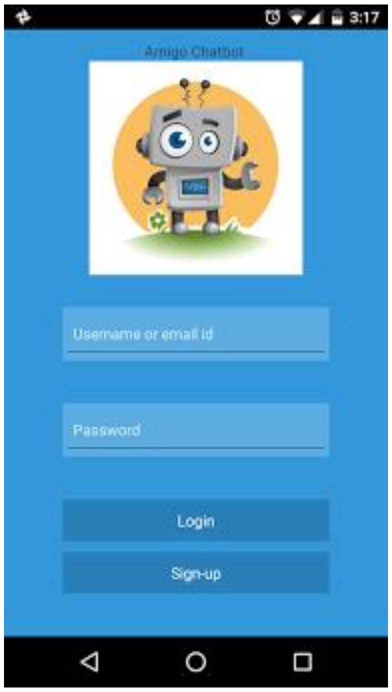


**Figure 33 - Login screen**

**Figure 34 - Login failure**

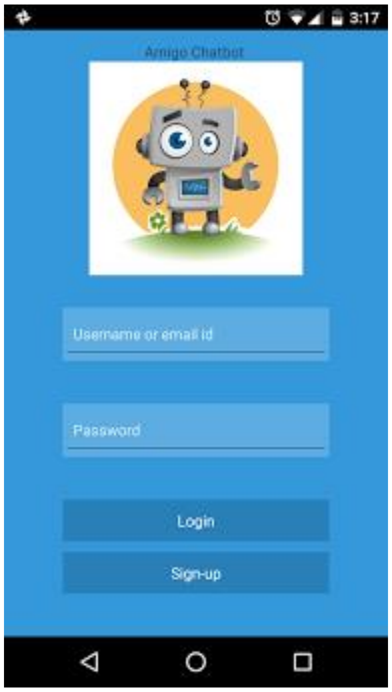


**Figure 33- Login Screen**

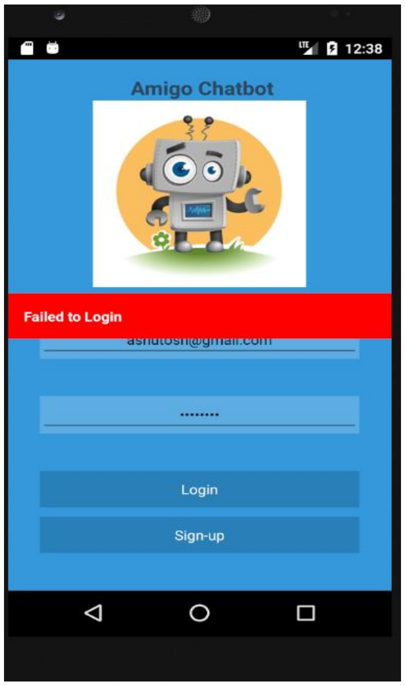


**Figure 34- Login Failed Screen**

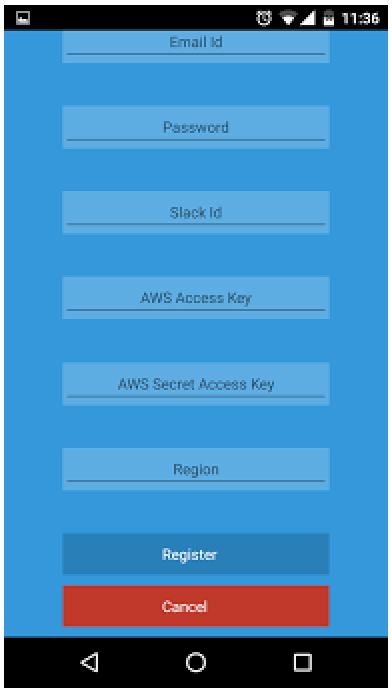
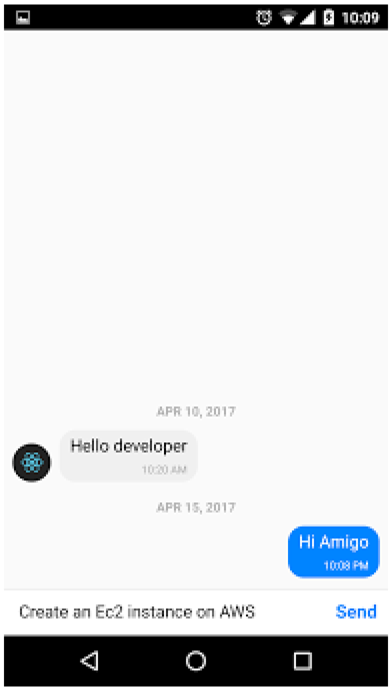


**Figure 35- Register Screen**



Hello developer
10:20 AM

Hi Amigo
10:08 PM

Create an Ec2 instance on AWS

**Figure 36 – Chat Room Screen**

## 5.2 MIDDLE-TIER IMPLEMENTATION

The middle-tier is made up of the following micro services deployed as Docker containers within a Docker Swarm cluster.

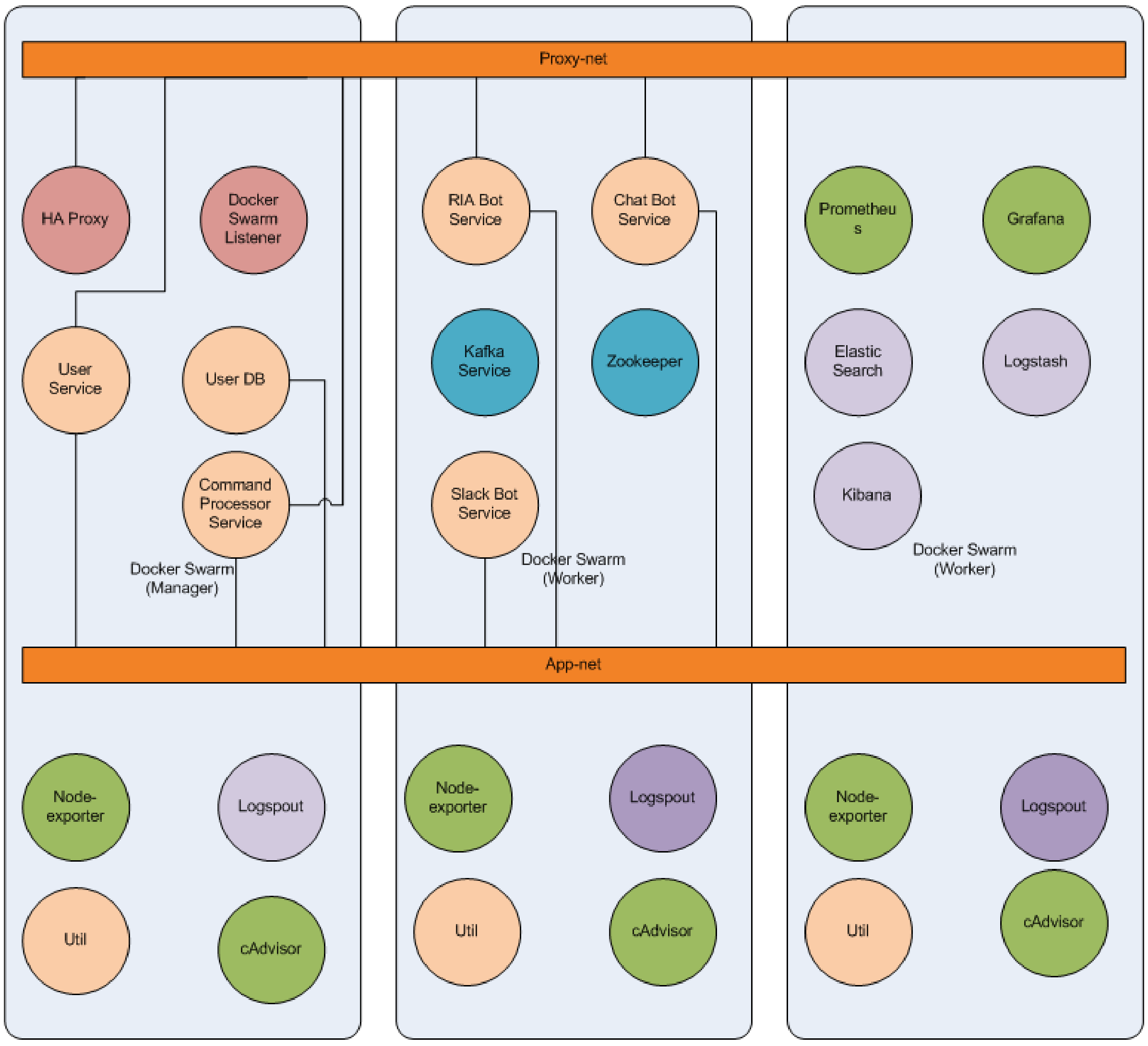


**Figure 37 - Docker Swarm Cluster running all Microservices**

Below table shows the various microservices and the legends (color) used in the above diagram to identify them:

| **Category** | **Microservice** | **Legend** | **Purpose** |
|---|---|---|---|
| **Reverse Proxy or API Gateway** | HA Proxy or Docker Flow Proxy [27] | Red | Reverse proxy or API gateway the routes the incoming messages based on the URL pattern. |
| | Docker Flow Swarm Listener [28] | | Auto registers any changes in containers IP with the HA Proxy service using its REST endpoint. |
| **Logging** | Logspout [17] | Purple | Logspout is a log router for Docker containers that runs inside Docker. It attaches to all containers on a host, then routes their logs wherever you want. It also has an extensible module system. It's a mostly stateless log appliance. It captures container logs written to stdout and stderr. |

| | | | |
|---|---|---|---|
| | Elasticsearch [14] | | Elastic search is a highly-scalable open-source full-text search and analytics engine. It allows you to store, search, and analyze big volumes of data quickly and in near real time. |
| | Logstash [15] | | Logstash is part of the Elastic Stack along with Beats, Elasticsearch and Kibana.<br>It is used to collect, aggregate, and parse your data, and then have Logstash feed this data into Elasticsearch. |
| | Kibana [16] | | Kibana is an open source analytics and visualization platform designed to work with Elasticsearch. Kibana is |

| | | | |
|---|---|---|---|
| | | | used to search, view, and interact with data stored in the Elasticsearch indices. One can easily perform advanced data analytics and visualize data in a variety of charts, tables and maps. |
| **Monitoring** | Prometheus [18] | Green | Prometheus is an open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach. |
| | Node-Exporter [20] | | Node exporter is exporter of hardware and OS metrics exposed by *NIX kernels to Prometheus. It is designed to monitor the host system. |

| | | | |
|---|---|---|---|
| | | | |
| | cAdvisor [21] | | cAdvisor (container advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. |
| | Grafana [19] | | Grafana is open source software for time series analytics and visualization. |
| **Amigo Chatbot Services** | User Service | Orange | Microservice for user profile CRUD. |
| | User DB | | Persistent store for user profile data. |
| | Command Processor Service | | Microservice for command processing on cloud provider. |
| | Command DB | | Persistent store for command to be executed, result of the execution, etc. |

| | | | |
|---|---|---|---|
| | RIABot Service | | Adapter service for Raspberry Pi Intelligent Assistant. |
| | SlackBot Service | | Adapter service for incoming messages from slack messenger. |
| | ChatBot Service | | Processes all incoming messages from different client types by inferring the intent from the message and then delegates to the command processor service for execution. |
| **Diagnostics Services** | Util | | It runs an instance of Alpine Linux and runs on each node of Swarm cluster. It is helpful to attach to this service and run the bash terminal and install diagnostics tools like drill or curl and triage an issue when the need arises. |

| | | | |
|---|---|---|---|
| | Docker Swarm Visualizer [35] | Not Shown | A visualization service that draws an image of the nodes in the swarm cluster and the services they are running. |

### 5.2.1 User Service

This micro service maintains the user profile data and provides the following endpoints for other services to interface with it:

a. User registration with the system.
b. User authentication and session management
c. User profile query
d. User account deletion.

The database used by user service is MongoDB.

### 5.2.2 Slack-Bot Service

This micro service listens to all incoming messages from slack service where the amigo chatbot is a participant. It could be either a direct message to amigo chatbot or a message in the channel with several other bots and users where amigo chatbot is also one of the participants. The listener parses the messages and selects only those that are addressed specifically to amigo chatbot.

Once a message addressed to amigo chatbot is selected, the listener invokes the wit.ai service to perform the Natural Language Processing (NLP) on the message and get the intent from the message.

If wit.ai returns the intent successfully then message is published to the Kafka message queue. If it fails to get the intent from the message, then the message is dropped and an error message is returned to the slack messenger user.

### 5.2.3 WIT.AI SERVICE

This is a third-party service that consumes the message and derives the intent of the message. Once the intent of the message can be derived successfully the chatbot service can process the message further.

### 5.2.4 COMMAND PROCESSOR SERVICE

This microservice processes the intent of the message by consuming it from the Kafka message queue topic. It then looks up its command DB for a matching intent and cloud provider and gets the corresponding Docker image name to use and the command to execute against that Docker image from the DB. It then pulls the Docker image (if not already present) or else the local cached image will be used for executing the command. The command is executed on the cloud provider and the corresponding response is returned to the user's client. MongoDB is used to implement the command DB.

## 5.3 DATA-TIER IMPLEMENTATION

### 5.3.1 USER DB

This DB is used for storing the user profile information. A user needs to create an account with the amigo chatbot’s user service either using its REST APIs or through the Web UI. Once an account exists, a Web UI or mobile App user can be authenticated.

Please note that a slack messenger user need not create an account with amigo chatbot as that user is already authenticated through slack messenger service.

### 5.3.2 COMMAND DB

This DB is used to store the mapping of an intent of message for a cloud provider to the Docker image and command that can be executed. The intent of the message and the cloud provider name can be used to lookup the Docker image and the command to execute. For each intent and cloud provider pair, a unique command needs to exist. The Docker image in most cases will be same for all commands to be executed for a given cloud provider.

## 5.4 IMPLEMENTATION PLAN

The project is being implemented using Agile methodology. For Continuous integration, Shippable and Travis CI services are used which does a build and test upon every commit to the GitHub repository (https://github.com/sjsucohort6/amigo-chatbot ) for the project.

Below table shows the last 2 sprints worth of tasks and their status:

| **Sprint 1 – Feb 11 – Feb 25** | | |
|---|---|---|
| **Tasks** | **Status** | **Completed by** |
| Work on wit.ai service integration | Completed | Swetha |
| Kafka with docker setup | Completed | Chetan |
| Command Processor Module | Completed | Watsh |
| Build docker image with awscli pre-installed | Completed | Ashutosh |
| **Sprint 2 – Feb 26 to March 12th** | | |
| Work on slack bot service and integrate with wit.ai | In Progress | Swetha |
| Work on Web UI – implement user sign-up | In Progress | Chetan |
| User service implementation | In Progress | Watsh |
| Mobile App UI | In Progress | Ashutosh |

# Chapter 6. Testing and Verification

## 6.1 INTRODUCTION

### 6.1.1 Purpose

This Test Plan document covers the necessary information required to effectively track and define the approaches that will drive the testing of the Amigo Chabot. The documents introduce:

- Test Strategy: Criteria on which test will be based on e.g.: objectives, assumptions, dates etc.; description of the process to perform valid test e.g.: creation of test cases, Schedule, Task to perform
- Execution Strategy: how to test needs to be performed and processed – reporting issues, implementing fixes
- Test Management: handling the logistics of the test and tasks that come up during execution e.g.: escalation procedures, communications

### 6.1.2 Project Overview

Amigo is an assistant tool to manage cloud operations. This tool allows user to provide the operation he/she would like to perform in natural language text. Tool gets the intent of the text message and maps it and calls required rest service to perform the task and gets the result back to user.

The functionality of the system allows users some sets of operations (which grows gradually) in the chat room. All the operations are subject to user's defined security policy where he/she can only run/performs commands he/she is authorized to.

### 6.1.3 AUDIENCE

- Project team members perform tasks specified in this document, and provide input and recommendations on this document.
- Project Lead Plans for the testing activities in the overall project schedule, reviews the document, tracks the performance of the test according to the task herein specified, approves the document and is accountable for the results.

## 6.2 TEST STRATEGY

### 6.2.1 TEST OBJECTIVES

The main objective of the test is to verify that the overall functionality of Amigo Chabot. As part of this we need to test functionality of the following modules

- Chatbot Service backend
- Bot Engine
- Mobile App
- Web Application
- Slack Integration

- Raspberry Pi Virtual Assistant

This test covers executing and verifying critical, high and medium severity defects with high priority and lower severity ones are prioritized for future fixing.

### 6.2.2 TEST ASSUMPTIONS

**Key Assumptions**

- Set of supported operations by the system are available to start this Functional Testing
- In each testing phase, number of cycles to be initiated depends on the defect rate of previous cycle. Only if there is high defect rate in cycle n-1, cycle n will be initiated.

**General**

- Performance testing is not considered for this.
- Exploratory testing will be carried out once build/module is ready for testing.
- Test environment and preparation activities will be taken care by corresponding module developer.
- All the test cases are design will be performed respective QA members of the module.
- System/Module will be treated as a black box; if the information/result is as expected, it will be assumed that the DB is working correctly.

### 6.2.3 TEST PRINCIPLES

- Testing will be focused on meeting quality

- Testing process will be well defined with the ability to change as needed
- All the test activities will depend on previous stages to avoid redundant effort
- Testing will be defined into phases with well-defined goals and objectives
- There will be criteria for entrance and exit

### 6.2.4 Scope and Levels of Testing

**Exploratory**
PURPOSE: to make critical defects fixed or removed before moving to next level of testing.

SCOPE: First level navigation

TESTERS: developer and testing team

METHOD: Carried without any test scripts and documentation

TIMING: at the beginning of each cycle

**Functional Test**
PURPOSE: will be performed to check functions of the application. It is performed against given set of inputs and validation of the system output.

SCOPE: below is the scope of the functional testing

- Chatbot Service backend
- Bot Engine
- Mobile App
- Web Application
- Slack Integration

- Raspberry Pi Virtual Assistant

TESTERS: Testing team

TIMING: after exploratory test is completed

TEST ACCEPTANCE CRITERIA

1. Approved functional specification document, Use-case document must be provided prior to start testing
2. Test cases needs to be approved
3. Development completed with unit tested features are eligible and the results of the unit test needs to be shared to avoid duplicate efforts.

**TEST DELIVERABLES**

**Table 2 - Test Deliverables**

| S.NO | Deliverable Name | Author | Reviewer |
|---|---|---|---|
| 1. | Test Plan | Test Lead | Project Manager |
| 2. | Functional Test Cases | Test Team | Project Manger |
| 3. | Logging Defects | Test Team | Test Lead |
| 4. | Daily/Weekly status report | Test team | Test Lead/Project Manager |
| 5. | Test Closure report | Test Lead | Project Manager |

**User Acceptance Test (UAT)**

PURPOSE: will be performed to validate the business logic. One final review of the complete system prior to delivery

TESTERS: is performed by the end users

METHOD: Test team writes some UAT test cases and those cases are validated probably not using scripts

TIMING: After exploratory and functional testing are completed. After completing this testing product is ready to be released/delivered

**TEST DELIVERABLES:**

**Table 3 - User Acceptance Test Deliverables**

| S. No. | Deliverable Name | Author |
|---|---|---|
| 1. | UAT Test cases | Test team |

## 6.2.5 Execution Strategy

**Entry and Exit Criteria**

The entry criteria refer to the desirable condition to start the test execution. The exit criteria refer to the conditions that needs to be met to proceed with implementation. Entry and exit criteria are flexible benchmarks.

**Table 4 - Entry and exit criteria**

| Exit Criteria | Test Team | Notes |
|---|---|---|
| 100% test scripts executed | | |

| | | |
|---|---|---|
| 95% pass rate | | |
| No open High/medium severity criteria | | |
| | | |

### 6.2.6 Test Cycles

- Functional testing will be carried out in two cycles.
- Aim of first cycle is to discover any blocking, critical/high defects.
- Aim of the second cycle is to identify remaining high and medium defects
- UAT consists of only one cycle

## 6.3 VALIDATION AND DEFECT MANAGEMENT

Testers are expected to run all the test cases in each of the cycles. Tester can log and report the issues using xlsx sheet shared via google drive and dev team can review them and work on fixing them. Testers/Manager are responsible for assigning the severity of the defect.

**Table 5 - Bug Severity**

| **Severity** | **Effect** |
|---|---|
| Critical (1) | Bug is critical can crash the system, potential data loss, hangs system |
| High (2) | Bug in vital component functionality |

| | |
|---|---|
| Medium (3) | Has workaround and could bring down the quality of the system |
| Low | Minimum impact in the product use |
| Cosmetic | No impact in the product use |

## 6.4 TEST MANAGEMENT PROCESS

### 6.4.1 Test Design Process

- Tester needs to understand each requirement and create test case to cover all requirements
- Each test case needs to be mapped to use cases to requirements (Traceability Matrix)
- Test cases will go through peer review and testers will rework on incorporating those comments.

### 6.4.2 Test Cases

Below is the list of some of the test Cases which needs to be covered during testing:

| **Test ID** | **Test Case** | **Steps** | **Expected Result** |
|---|---|---|---|
| **US001** | User Registration | Signup using REST API by doing POST /api/v1.0/users endpoint. | 201 Created response should be returned. User |

| | | | |
|---|---|---|---|
| | | | should be created in user DB. |
| **US002** | Get all users | Fetch all users using REST API by doing GET on /api/v1.0/users endpoint. | 200 OK response should be returned. |
| **US003** | Get a user | Get user by ID using REST API by doing GET on /api/v1.0/users/{userId} endpoint. | 200 OK response should be returned. |
| **US004** | Update a user profile | Update a user's profile by doing PUT on /api/v1.0/users/{userId} endpoint. | 200 OK response should be returned and updated user entity should be returned. |
| **US005** | Delete a user | Delete a user by doing DELETE on /api/v1.0/users/{userId}. | 200 OK response returned. |
| **US006** | Web UI Tests | Repeat tests US001 through to US005 with Web UI | |
| **US007** | Mobile UI Tests | Repeat tests US001 through to US005 with Mobile UI | |

| | | | |
|---|---|---|---|
| **SL008** | Slack user send message to amigo bot | Send a message to amigo bot from slack UI for a user who is registered with amigo bot system. | Bot service should respond. |
| **SL009** | Send message that can be parsed for intent. | Send message “aws list ec2 instances” | Bot should return the result of command execution. |
| **SL010** | Send message that is known that it cannot be parsed for intent, like missing cloud provider name. | Send message “list ec2 instances” | Bot should return failure message. |
| **SL011** | Send message that is correct but known that it cannot be parsed for intent. | Send message “aws iam list-users” | Bot should return correct response still. |
| **SL012** | Send a message that is not correct by itself | Send message “list iam users aws” | Bot should return aws CLI help. |

| | | | |
|---|---|---|---|
| | and does not have intent mapping. | | |
| **RI013** | Send a correct message that has intent mapping. | Send message “list aws ec2 instances” | Bot should return success message and blink(1) light should glow GREEN. |
| **RI014** | Send an incorrect message that has no intent mapping. | Send message “list my instances” | Bot should return failure message and blink(1) should glow RED. |
| **RI015** | Send a message that has no intent mapping but is correct. | Send message “aws list ec2 instances” | Bot should return success message and blink(1) should glow GREEN. |

## 6.5 PROJECT MANAGEMENT

Project Manager: Reviews test plan, test strategy and test estimates.

### 6.5.1 Test Planning (Test Lead)

- Creates test plan, test cases and expected results and execution script
- Manages defect
- Communicates with the development team

### 6.5.2 Test Team

- Performs execution and validation
- Identifies and assigns priority to defects
- Prepares testing metrics

### 6.5.3 Development Team

- Keeps test team and project team informed with feature delivery date
- Responsible for developing assigned components / features
- Fixes the defects according to the schedule
- Helps test team in validating of the results (if required)

# Chapter 7. PERFORMANCE AND BENCHMARKS

## 7.1 PERFORMANCE METRICS

The system’s performance is calculated on various factors, some are as follows:

- **Response time**: Response time is the time system will take to respond to the user’s request. Our system design ensures minimal response time.
- **Processing time:** When user gives the input to Amigo to process some request, the time taken to process the request is the processing time. The processing time should be minimum.
- **Throughput:** This is the most important parameter from user’s perspective. It is the number of requests a system can process in given time. If the throughput exceeds than expected, there is a possibility of Amigo chatbot server becoming non-responsive.
- **Query and reporting time:** After processing inputs from user, system will query and respond to the user with the respective output. This whole time is calculated as query and reporting time. Our system has optimal query and reporting time.

In addition to the above, we will be capturing 5 important metrics specific to chatbot:

- **Active and engaged users:** It is the number of active sessions of a user w.r.t the number of total sessions with that user, here the user reads a message sent by the bot. Engaged rate is the number of engaged sessions per total sessions, here the user responds to a message. We can find out the frequently typed messages from the user and can adapt the bot accordingly.

- **Confusion Triggers:** It is the situation where chatbot doesn't understand how to process the user's request. If we understand what user inputs are causing this issue and help to optimize the programming model.
- **Conversation Steps:** It is the count of exchanges between Amigo chatbot and the user. This will help us to understand the average number of conversation steps between user and the chatbot. The length of conversations (either short or long than the average) can help us diagnose any problem related to the chatbot performance.
- **Average number of conversations per user:** This parameter tells us if there is an issue with the chatbot service.
- **Retention Rate:** This parameter tells us whether the users are coming back to the chatbot service and if they are not then we can make changes in our chatbot service accordingly.

## 7.2 PERFORMANCE TOOLS

Each Amigo chatbot service has been exposed as an API, hence every API endpoint like create user, update user, delete user etc. will be tested for single as well as multiple concurrent threads to get the performance footprint. In addition to the above-mentioned parameters, we will also be capturing the CPU and Memory utilization for each test. This will give us a better picture on how and when to scale our environment.

### 7.2.1 Apache JMeter

We use Apache JMeter to carry out performance tests. JMeter uses a concept called Test Plan that has number of controllable parameters like Thread Group, Timer, and Sampler etc. Every test plan is one performance scenario. Following is the explanation of some concepts:

**Thread Group:** It is the beginning step of every test and consists of controllers and samplers. In simple terms, it is the parameter to control the number of threads you require to execute the test. Additionally, you can also control the ramp up time which is the time until all the threads will start executing concurrently to make the simulation represent real-life scenario.

**Sampler:** It tells JMeter to send the requests to the server. There are number of samplers like FTP Request, HTTP Request, and JDBC Request etc.

## 7.3 PERFORMANCE TESTS

We test all the API endpoints related to Amigo chat service. To start with, we consider all the user related APIs.

### 7.2.2 Machine Details

The performance is tested on a machine with following configurations:

| Processor Name | Intel Core i7 |
|---|---|
| Processor Speed | 2.8 GHz |
| Number of Processors | 1 |
| Total Number of Cores | 4 |
| L2 Cache (per Core) | 256 KB |
| L3 Cache | 6 MB |
| Memory | 16 GB |

### *7.2.3* Performance Results

Each test is carried out for a duration on 2 minutes.

**POST /api/v1.0/signup**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 7082 | 16 | 3 | 2002 | 58.2 | 10 | ~0 |
| 5 | 7248 | 78 | 4 | 2014 | 59.1 | 15 | ~0 |
| 10 | 7618 | 145 | 4 | 2056 | 62.2 | 20 | ~0 |
| 20 | 8190 | 285 | 4 | 2033 | 66.8 | 30 | ~0 |

**POST /api/v1.0/login**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 1701 | 70 | 65 | 104 | 14.2 | 2 | ~0 |
| 5 | 5868 | 101 | 66 | 192 | 48.9 | 9 | ~0 |
| 10 | 4880 | 245 | 66 | 308 | 40.6 | 10 | ~0 |

**POST /api/v1.0/users**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 1690 | 69 | 63 | 100 | 13.5 | 2 | ~0 |
| 5 | 5762 | 98 | 64 | 185 | 47.5 | 9 | ~0 |
| 10 | 4758 | 234 | 64 | 298 | 39.8 | 10 | ~0 |

**GET /api/v1.0/users**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 7091 | 17 | 3 | 2002 | 58.5 | 10 | ~0 |
| 5 | 7251 | 83 | 4 | 2009 | 59.7 | 15 | ~0 |
| 10 | 7624 | 158 | 4 | 2034 | 62.6 | 20 | ~0 |
| 20 | 8206 | 292 | 4 | 2027 | 67.3 | 30 | ~0 |

**GET /api/v1.0/users/{userId}**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 7073 | 16 | 3 | 2001 | 58.5 | 10 | ~0 |
| 5 | 7245 | 78 | 4 | 2005 | 59.6 | 15 | ~0 |
| 10 | 7598 | 151 | 4 | 2029 | 62.2 | 20 | ~0 |
| 20 | 8197 | 289 | 4 | 2023 | 66.7 | 30 | ~0 |

**HEAD /api/v1.0/users/{userId}**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 7082 | 16 | 4 | 2002 | 58.7 | 10 | ~0 |
| 5 | 7245 | 80 | 4 | 2007 | 59.8 | 15 | ~0 |
| 10 | 7607 | 154 | 4 | 2030 | 61.9 | 20 | ~0 |
| 20 | 8202 | 290 | 4 | 2025 | 67.7 | 30 | ~0 |

**PUT /api/v1.0/users/{userId}**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 7085 | 17 | 3 | 1995 | 58.7 | 10 | ~0 |
| 5 | 7179 | 79 | 4 | 1989 | 59.5 | 15 | ~0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 7595 | 145 | 4 | 2034 | 62.3 | 20 | ~0 |
| 20 | 8194 | 285 | 4 | 2027 | 66.4 | 30 | ~0 |

**DELETE /api/v1.0/users/{userId}**

| # of Threads | # of Samples | Average | Min | Max | Throughput (request/sec) | % CPU Utilization | Memory Usage (KB) |
|---|---|---|---|---|---|---|---|
| 1 | 7084 | 17 | 3 | 2002 | 58.2 | 10 | ~0 |
| 5 | 7248 | 74 | 4 | 1995 | 58.7 | 15 | ~0 |
| 10 | 7605 | 144 | 4 | 2012 | 63.5 | 20 | ~0 |
| 20 | 8195 | 285 | 4 | 2003 | 65.6 | 30 | ~0 |

# Chapter 8. DEPLOYMENT, OPERATIONS, MAINTENANCE

## 8.1 DEPLOYMENT

For deployment, Docker Swarm mode is used for the project.

In the development environment, the swarm cluster can be run locally using virtualbox driver as:

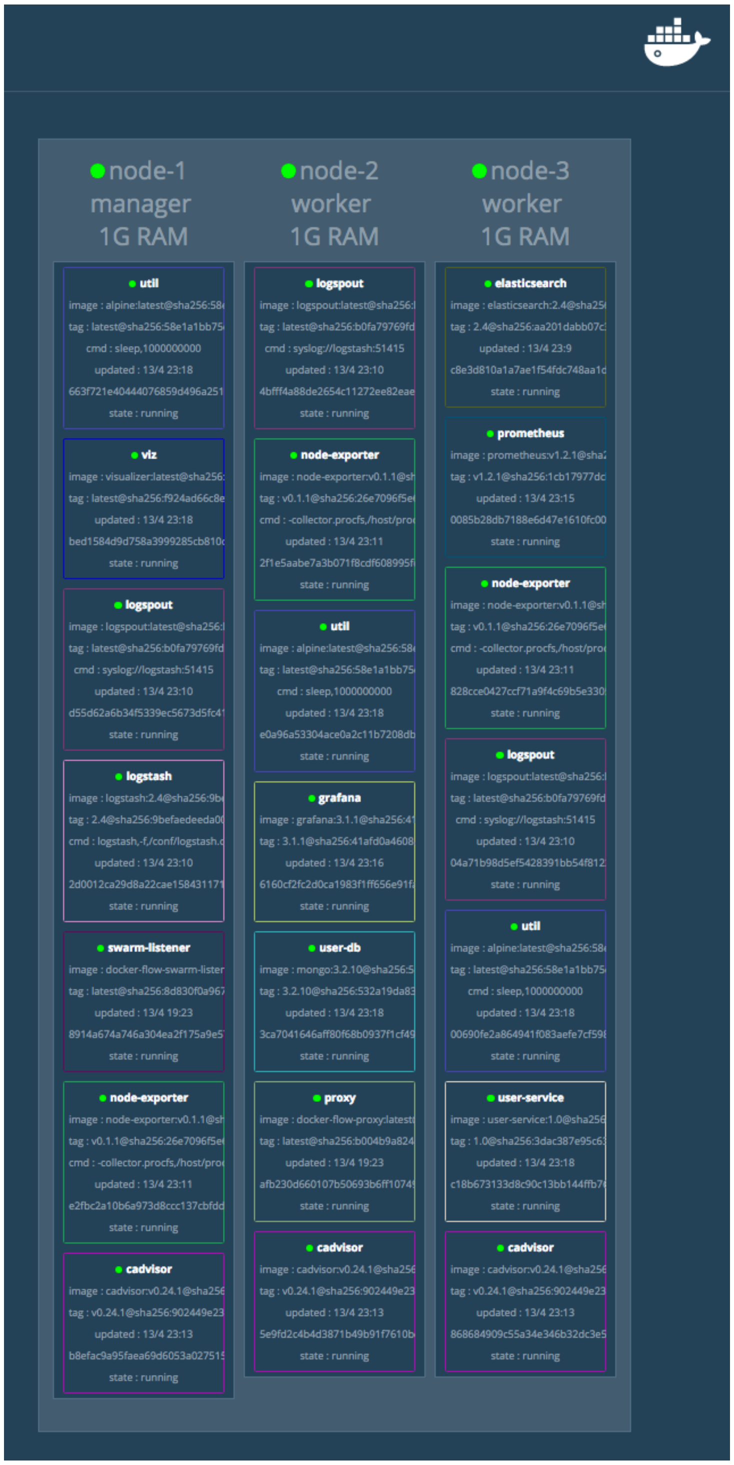
**Table 6 Docker Swarm Cluster Creation**

```
#!/usr/bin/env bash
# Create swarm cluster nodes with virtualbox
for i in 1 2 3; do
  docker-machine create -d virtualbox node-$i
done


# Make node1 swarm manager
eval $(docker-machine env node-1)

docker swarm init \
  --advertise-addr $(docker-machine ip node-1)


# Get the token to join worker nodes
TOKEN=$(docker swarm join-token -q worker)

# Make nodes 2 and 3 as swarm workers
for i in 2 3; do
  eval $(docker-machine env node-$i)

  docker swarm join \
    --token $TOKEN \
    --advertise-addr $(docker-machine ip node-$i) \
    $(docker-machine ip node-1):2377
done

echo "Created swarm cluster"
```

Once the cluster is created, each service is added using docker create service command as shown below:

```
eval $(docker-machine env node-1)

# Create user-db service
docker service create --name user-db \
    --network user-net \
    --log-driver=gelf \
    --log-opt gelf-address=udp://127.0.0.1:12201 \
    --log-opt tag="user-db" \
    mongo:3.2.10

# Create user-service service
docker service create --name user-service \
  -e DB=user-db \
  --network user-net \
  --network proxy-net \
  --label com.df.notify=true \
  --label com.df.distribute=true \
  --label com.df.servicePath=/api/v1.0/users \
  --label com.df.port=8080 \
  --log-driver=gelf \
  --log-opt gelf-address=udp://127.0.0.1:12201 \
  --log-opt tag="user-service" \
  sjsucohort6/user-service:1.0

docker service ps user-db
docker service ps user-service

echo "Created user service and user db"
```

Docker networking is used to connect the services by making them join on the same overlay network then each service can call the other by its name.



● node-1
manager
1G RAM

● util
image : alpine:latest@sha256:58
tag : latest@sha256:58e1a1bb75
cmd : sleep,1000000000
updated : 13/4 23:18
663f721e40444076859d496a251
state : running

● viz
image : visualizer:latest@sha256:
tag : latest@sha256:f924ad66c8e
updated : 13/4 23:18
bed1584d9d758a3999285cb810
state : running

● logspout
image : logspout:latest@sha256:
tag : latest@sha256:b0fa79769fd
cmd : syslog://logstash:51415
updated : 13/4 23:10
d55d62a6b34f5339ec5673d5fc4
state : running

● logstash
image : logstash:2.4@sha256:9b
tag : 2.4@sha256:9befaedeeda0
cmd : logstash,-f,/conf/logstash.
updated : 13/4 23:10
2d0012ca29d8a22cae15843117
state : running

● swarm-listener
image : docker-flow-swarm-lister
tag : latest@sha256:8d830f0a967
updated : 13/4 19:23
8914a674a746a304ea2f175a9e5
state : running

● node-exporter
image : node-exporter:v0.1.1@sh
tag : v0.1.1@sha256:26e7096f5e
cmd : -collector.procfs,/host/pro
updated : 13/4 23:11
e2fbc2a10b6a973d8ccc137cbfdd
state : running

● cadvisor
image : cadvisor:v0.24.1@sha256
tag : v0.24.1@sha256:902449e23
updated : 13/4 23:13
b8efac9a95faea69d6053a02751
state : running

● node-2
worker
1G RAM

● logspout
image : logspout:latest@sha256:
tag : latest@sha256:b0fa79769fd
cmd : syslog://logstash:51415
updated : 13/4 23:10
4bfff4a88de2654c11272ee82eae
state : running

● node-exporter
image : node-exporter:v0.1.1@sh
tag : v0.1.1@sha256:26e7096f5e
cmd : -collector.procfs,/host/pro
updated : 13/4 23:11
2f1e5aabe7a3b071f8cdf608995f
state : running

● util
image : alpine:latest@sha256:58
tag : latest@sha256:58e1a1bb75
cmd : sleep,1000000000
updated : 13/4 23:18
e0a96a53304ace0a2c11b7208db
state : running

● grafana
image : grafana:3.1.1@sha256:41
tag : 3.1.1@sha256:41afd0a4608
updated : 13/4 23:16
6160cf2fc2d0ca1983f1ff656e91f
state : running

● user-db
image : mongo:3.2.10@sha256:5
tag : 3.2.10@sha256:532a19da83
updated : 13/4 23:18
3ca7041646aff80f68b0937f1cf49
state : running

● proxy
image : docker-flow-proxy:latest
tag : latest@sha256:b004b9a824
updated : 13/4 19:23
afb230d660107b50693b6ff1074
state : running

● cadvisor
image : cadvisor:v0.24.1@sha256
tag : v0.24.1@sha256:902449e23
updated : 13/4 23:13
5e9fd2c4b4d3871b49b91f7610b
state : running

● node-3
worker
1G RAM

● elasticsearch
image : elasticsearch:2.4@sha25
tag : 2.4@sha256:aa201dabb07c
updated : 13/4 23:9
c8e3d810a1a7ae1f54fdc748aa1d
state : running

● prometheus
image : prometheus:v1.2.1@sha
tag : v1.2.1@sha256:1cb17977dc
updated : 13/4 23:15
0085b28db7188e6d47e1610fc00
state : running

● node-exporter
image : node-exporter:v0.1.1@sh
tag : v0.1.1@sha256:26e7096f5e
cmd : -collector.procfs,/host/pro
updated : 13/4 23:11
828cce0427ccf71a9f4c69b5e330
state : running

● logspout
image : logspout:latest@sha256:
tag : latest@sha256:b0fa79769fd
cmd : syslog://logstash:51415
updated : 13/4 23:10
04a71b98d5ef5428391bb54f812
state : running

● util
image : alpine:latest@sha256:58
tag : latest@sha256:58e1a1bb75
cmd : sleep,1000000000
updated : 13/4 23:18
00690fe2a864941f083aefe7cf59
state : running

● user-service
image : user-service:1.0@sha256
tag : 1.0@sha256:3dac387e95c6
updated : 13/4 23:18
c18b673133d8c90c13bb144ffb7
state : running

● cadvisor
image : cadvisor:v0.24.1@sha256
tag : v0.24.1@sha256:902449e23
updated : 13/4 23:13
868684909c55a34e346b32dc3e5
state : running

**Figure 35 - Docker Swarm Visualizer**

Docker Swarm Visualizer showing 3 nodes of the cluster running microservices across all nodes.

In production, Docker Swarm cluster will be deployed on Amazon Web Services EC2 instances with Docker engine.

## 8.2 OPERATION AND MAINTENANCE

To ensure that application is running successful and fulfilling its all requirements and features. Application operation phase continues till the application life. All the operation and maintenance of our project will be done via Kubernetes. Which will monitor system performance and efficiency.

## 8.3 DEPLOYMENT PLAN

**Table 7 – Deployment Plan**

| **Phase** | **Plan** | **Status** |
|---|---|---|
| Setting up programming and execution environment | 1. Install Slack<br>2. Create wit.ai developer account.also<br>3.Setup Docker | Completed |
| Set up Development Environment | 1. Set up Slack chatbot<br>2. Set up IDE for Development<br>3. Setup Mongo DB<br>4.setting up Wit.ai instance | Completed |

| | | |
|---|---|---|
| | 5. Setting up React native for mobile development<br>6.Setting up Kafka Clusters | |
| Executing programs and hardware simulations | 1. Integrate wit.ai with chatbot service<br>2.Integrate Docker hub to pull AWS - cli image<br>3. Develop test cases for each module. | Completed |
| Implementing Modules | 1. Wit.ai service to find intent<br>2. Kafka message queue<br>3. Command processor Module<br>4. Client UI Module<br>5. Mobile Application | Completed |
| Integrating Module | 1. Integrate all module and test them.<br>2. Test Module in development environment.<br>3. Test all use cases. | Completed |

| | | |
|---|---|---|
| | | |

# Chapter 9. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

## 9.1 SUMMARY

In this project, “Amigo: A Chatbot for cloud ops management” we have presented and implemented a tool to assist humans to preform cloud operations by taking conversational text as input. This bot is built with serverless framework using AWS Lambda. In depth architecture details of Amigo including middle, client and data tier design are discussed and shows how the bot is scalable. Amigo has been trained with few basic set of questions related to the small set of the operations (AWS commands). As part of this project maintenance, we aim at training our model to support wider set of operations and cloud platforms. This new approach to communication that allows teams to collaborate and manage various aspects of their infrastructure. It enables improving how IT teams collaborate to handle DevOps by making it more visible, efficient and simple.

## 9.2 CONCLUSIONS

Bots can help the team to be more productive and to accomplish tasks with more ease. From our studies above, it can be said that there are various approaches and methods used in Chatbot design. One must note that all these techniques of design are still a matter for debate and no common approach has been identified yet. Amigo’s architecture is fully server less, secure and responsive. It allows IT teams to collaborate and manage various aspects of their infrastructure.

There is still an opportunity to add few more integrations with enterprise solutions and thereby making it extensible platform.

## 9.3 RECOMMENDATIONS FOR FURTHER RESEARCH

With the advances in Artificial Intelligence combined with rising popularity of mobile chatting apps throws a new wave of innovation in digital commerce which is more conversational and personal in nature. Technology has also advanced to a point where voice digital assistants are beginning to become useful for the average consumer. This has led to a new trend – “conversational commerce” which began pickup speed in 2016.

Some companies like H&M are already using chatbots for marketing via messenger services like Kik. Aside from marketing, chatbots will continue to play a role in online commerce and of course customer service.

# GLOSSARY

*Source: Wikipedia*

**Table 8 – Glossary**

| **Term** | **Description** |
|---|---|
| **Amazon Echo** | Amazon Echo (known in-development as Doppler or Project D and shortened and referred to as Echo) is a smart speaker developed by Amazon.com. The device consists of a 9.25-inch (23.5 cm) tall cylinder speaker with a seven-piece microphone array. |
| **Amazon Web Services (AWS)** | Amazon Web Services, a subsidiary of Amazon.com, offers a suite of cloud-computing services that make up an on-demand computing platform. These services operate from 14 geographical regions across the world. |
| **Apache Kafka** | |
| **Apache Mesos** | Apache Mesos is an open-source cluster manager that was developed at the University of California, Berkeley. It "provides efficient resource isolation and sharing across distributed applications, or frameworks". The software |

| | |
|---|---|
| | enables resource sharing in a fine-grained manner, improving cluster utilization. |
| **Apache Spark** | Apache Spark is a fast, in-memory data processing engine with elegant and expressive development APIs to allow data workers to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets. |
| **Apache Zookeeper** | ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. |
| **Apache ZooKeeper** | |
| **Artificial Intelligence Markup Language (AIML)** | AIML (Artificial Intelligence Markup Language) is an XML-compliant language that's easy to learn, and makes it possible for you to begin customizing an Alicebot or creating one from scratch within minutes. The most |

| | |
|---|---|
| | important units of AIML are: <aiml>: the tag that begins and ends an AIML document. |
| **Cassandra** | Apache Cassandra is a free and open-source distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. |
| **Chat-Bot** | Short for **chat** ro**bot**, a computer program that simulates human conversation, or chat, through artificial intelligence. Typically, a chat bot will communicate with a real person, but applications are being developed in which two chat bots can communicate with each other. Chat bots are used in applications such as ecommerce customer service, call centers and Internet gaming. Chat bots used for these purposes are typically limited to conversations regarding a specialized purpose and not for the entire range of human communication.<br>One well known example of a chat bot is ALICE.<br>A chat bot is also called a chatterbot. |

| | |
|---|---|
| **Cloud Computing** | The practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server or a personal computer. |
| **Consul** | |
| **DevOps** | DevOps (a clipped compound of development and operations) is a term used to refer to a set of practices that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes. |
| **Docker** | Docker is an open-source project that automates the deployment of Linux applications inside software containers. Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Linux. |
| **Docker Hub** | Docker Hub is a cloud-based registry service which allows one to link to code repositories, build images and test them, stores manually pushed images, and links to the |

| | |
|---|---|
| | Docker Cloud so one can deploy images to hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration and workflow automation throughout the development pipeline. |
| **Docker Swarm** | Cluster management and orchestration features embedded in the Docker engine are built using **SwarmKit**. Docker engine participating in a cluster are running in swarm mode. |
| **Dropwizard** | Dropwizard is a Java framework for developing ops-friendly, high performance, RESTful webservices. Dropwizard pulls together stable, mature libraries from the Java ecosystem into a simple light-weight package. It has out-of-the-box support for configuration, application metrics, logging, operational tools and much more. |
| **Elasticsearch** | Elastic search is a highly-scalable open-source full-text search and analytics engine. It allows you to store, search, |

| | |
|---|---|
| | and analyze big volumes of data quickly and in near real time. |
| **Facebook Messenger** | Facebook Messenger (sometimes abbreviated as Messenger) is an instant messaging service and software application which provides text and voice communication. |
| **Google cAdvisor** | cAdvisor (container advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. |
| **Grafana** | Grafana is open source software for time series analytics and visualization. |
| **HTTP** | Hypertext transfer protocol is an application protocol for distributed, collaborative, and hypermedia information systems. HTTP is the foundation of data communication for the world wide web. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. |

| | |
|---|---|
| **Jersey** | Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation. |
| **Kibana** | Kibana is an open source analytics and visualization platform designed to work with Elasticsearch. Kibana is used to search, view, and interact with data stored in the Elasticsearch indices. One can easily perform advanced data analytics and visualize data in a variety of charts, tables and maps. |
| **Logspout** | Logspout is a log router for Docker containers that runs inside Docker. It attaches to all containers on a host, then routes their logs wherever you want. It also has an extensible module system. It's a mostly stateless log appliance. It captures container logs written to stdout and stderr. |

| | |
|---|---|
| **Logstash** | Logstash is part of the Elastic Stack along with Beats, Elasticsearch and Kibana.<br>It is used to collect, aggregate, and parse your data, and then have Logstash feed this data into Elasticsearch. |
| **Marathon** | Marathon is a production-grade container orchestration platform for Mesosphere's Datacenter Operating System (DC/OS) and Apache Mesos. |
| **Microsoft Azure** | Microsoft Azure /ˈæʒər/ is a cloud computing platform and infrastructure created by Microsoft for building, deploying, and managing applications and services through a global network of Microsoft-managed data centers. |
| **Natural Language Processing (NLP)** | Natural language processing is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages. As such, NLP is related to the area of human–computer interaction. |

| | |
|---|---|
| **Node-exporter** | Node exporter is exporter of hardware and OS metrics exposed by *NIX kernels to Prometheus. It is designed to monitor the host system. |
| **Prometheus** | Prometheus is an open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach. |
| **Quartz Job Scheduler** | Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application. It can be used to create simple or complex schedules for executing tens of thousands of jobs; jobs whose tasks are defined as standard Java components that may execute virtually any task. It includes many enterprise class features like support for JTA transactions and clustering. |
| **React JS** | A JavaScript library for building user interfaces |

| | |
|---|---|
| **React Native** | Building mobile apps with React that work same as native apps and uses same fundamental UI building blocks as regular iOS and Android apps using JavaScript and React. |
| **REST** | Representational State Transfer (REST) WebServices are one way of providing interoperability between computer systems on the internet. |
| **Slack** | Slack is a cloud-based team collaboration tool co-founded by Stewart Butterfield, Eric Costello, Cal Henderson, and Serguei Mourachov. Slack began as an internal tool used by their company, Tiny Speck, in the development of Glitch, a now defunct online game. |
| **Webhook** | A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST. A web application implementing WebHooks will POST a message to a URL when certain things happen. |

# REFERENCES

[1] J. Kreps, I Heart Logs, O'Reilly Media, Inc., 2014.

[2] Facebook, "React," Facebook, 2017. [Online]. Available: https://facebook.github.io/react/. [Accessed 15 April 2017].

[3] Facebook, "React Native," Facebook, [Online]. Available: https://facebook.github.io/react-native/.

[4] Slack, "Slack API Bot Users," Slack, [Online]. Available: https://api.slack.com/bot-users.

[5] T. Pant, Building a Virtual Assistant for Raspberry Pi: The practical guide for constructing a voice-controlled virtual assistant, 1st ed., Apress, 2016.

[6] Docker, "What is Docker," Docker, [Online]. Available: https://www.docker.com/what-docker.

[7] Docker, "Swarm mode key concepts," [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/.

[8] Docker, "Overview of Docker Hub," [Online]. Available: https://docs.docker.com/docker-hub/.

[9] Facebook, "wit.ai," [Online]. Available: https://wit.ai/.

[10] Hashicorp, "Introduction to Consul," [Online]. Available: https://www.consul.io/intro/.

[11] Apache, "Apache Kafka a distributed streaming platform," [Online]. Available: https://kafka.apache.org/.

[12] Apache, "Apache ZooKeeper," [Online]. Available: https://kafka.apache.org/.

[13] HAProxy, "HAProxy," [Online]. Available: http://www.haproxy.org/.

[14] Elasticsearch, "Elasticsearch Reference," [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html.

[15] Elasticsearch, "Logstash - transport and process your logs, events, or other data," [Online]. Available: https://github.com/elastic/logstash.

[16] Elasticsearch, "Kibana User Guide Introduction," [Online]. Available: https://www.elastic.co/guide/en/kibana/current/introduction.html.

[17] Gliderlabs, "Log routing for Docker container logs," [Online]. Available: https://github.com/gliderlabs/logspout.

[18] Prometheus, "Prometheus - Monitoring system & time series database," [Online]. Available: https://prometheus.io/.

[19] Grafana Labs, "Grafana," [Online]. Available: https://grafana.com/.

[20] Prometheus, "Node Exporter," [Online]. Available: https://github.com/prometheus/node_exporter.

[21] Google, "cAdvisor," [Online]. Available: https://github.com/google/cadvisor.

[22] Dropwizard Team, "Dropwizard," [Online]. Available: http://www.dropwizard.io/1.1.0/docs/.

[23] Software AG, "Quartz Job Scheduler," [Online]. Available: http://www.quartz-scheduler.org/.

[24] Oracle Corporation, "Jersey - RESTful WebServices in Java," [Online]. Available: https://jersey.java.net/.

[25] MongoDB, "Introduction to Mongo DB," [Online]. Available: https://docs.mongodb.com/manual/introduction/.

[26] A. Team, "Raspberry Pi Intelligent Assistant (RIA)," [Online]. Available: https://github.com/sjsucohort6/ria.

[27] V. Farcic, "Docker Flow Proxy," [Online]. Available: https://github.com/vfarcic/docker-flow-proxy.

[28] V. Farcic, "Docker Flow Swarm Listener," [Online]. Available: https://github.com/vfarcic/docker-flow-swarm-listener.

[29] blink(1), "The USB RGB LED notification light," [Online]. Available: https://blink1.thingm.com/.

[30] V. Farcic, "Microservices: The Essential Practices," [Online]. Available: https://technologyconversations.com/2015/11/10/microservices-the-essential-practices/.

[31] V. Badola, "Microservice architecture: advantages and drawbacks," Cloudacademy, 30 November 2015. [Online]. Available: http://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/. [Accessed 15 April 2017].

[32] C. Richardson, "Pattern: Microservice Architecture," [Online]. Available: http://microservices.io/patterns/microservices.html.

[33] C. Richardson. [Online]. Available: http://microservices.io/patterns/apigateway.html.

[34] C. Richardson. [Online]. Available: http://microservices.io/patterns/server-side-discovery.html.

[35] Docker, "Docker Swarm Visualizer," [Online]. Available: https://github.com/dockersamples/docker-swarm-visualizer.

[36] Turbonomic, "Getting started with Docker Swarm," [Online]. Available: https://turbonomic.com/blog/on-technology/getting-started-with-docker-swarm-part-1/.

[37] T. Pant, Building a Virtual Assistant for Raspberry Pi: The practical guide for constructing a voice-controlled virtual assistant, APress, 2016.

[38] S. Goasguen, Docker Cookbook, O'Reilly Media, Inc., 2015.

[39] K. Hightower, Kubernetes: Up and Running, O'Reilly Media, Inc., 2017.

[40] V. Farcic, The DevOps 2.0 Toolkit, Packt Publishing, 2016.

[41] B. Burke, RESTful Java with JAX-RS 2.0, 2nd Edition ed., O'Reilly Media, Inc., 2013.

[42] D. M. Beazley, Python Essential Reference, 4th Edition ed., Addison-Wesley Professional, 2009.

[43] K. Becker, Building Voice-Enabled Apps with Alexa, Bleeding Edge Press, 2017.

[44] WhatsBroadcast, "The Importance of Chatbot Metrics," [Online]. Available: https://www.whatsbroadcast.com/the-importance-of-chatbot-metrics/.

[45] Facebook, "wit.ai," [Online]. Available: https://wit.ai/.

# Appendices

## Appendix A. Apache Kafka

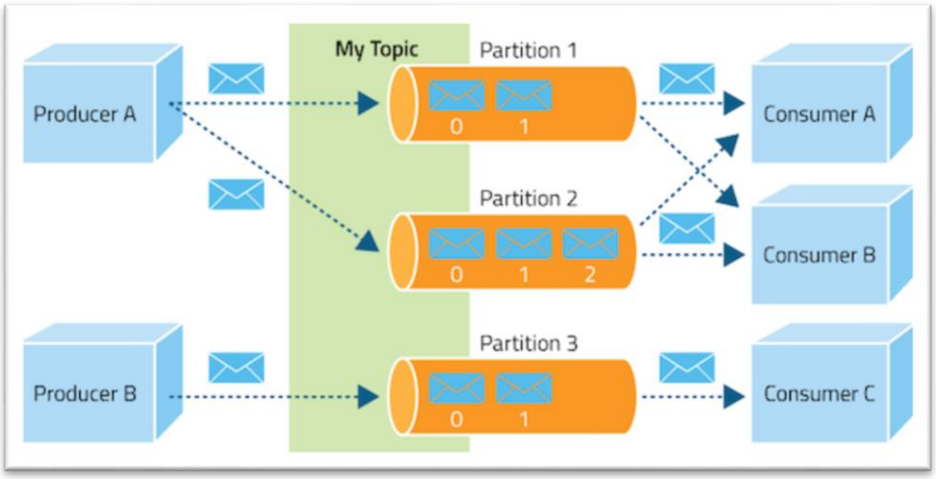


**Figure 16- Apache Kafka Architecture**

1. Message Oriented Middleware (MOM) such as Apache Qpid, Rabbit MQ, Microsoft MQ and IBM MQ Series were used for exchanging messages across various components. While these products are good at implementing the publisher/subscriber pattern (Pub/Sub), they are not specifically designed for dealing with large streams of data originating from thousands of publishers. Most of the MOM software have a broker that exposes Advanced Message Queuing Protocol (AMQP) protocol for asynchronous messaging.
2. Kafka is designed from the ground up to deal with millions of firehose-style events generated in rapid succession. It guarantees low-latency, “at-least-once”, delivery of messages to consumers. Kafka also supports retention of data for offline consumers, which means that the data can be processed either in real-time or in offline mode.

3. Kafka is designed to be a distributed commit log. Much like relational databases, it can provide a durable record of all transactions that can be played back to recover the state of a system.
4. Kafka provides redundancy, which ensures high availability of data even when one of the servers faces disruption.
5. Multiple event sources can concurrently send data to a Kafka cluster, which will reliably get delivered to multiple destinations.
6. Key concepts:
    - Message – Each message is a key/value pair. Irrespective of data type, Kafka always converts messages into byte arrays.
    - Producers – or publisher clients that produce data
    - Consumers – are subscribers or readers that read the data. Unlike subscribers in MOM, Kafka consumers are stateful, which means they are responsible for remembering the cursor position, which is called an offset. The consumer is also a client of Kafka cluster. Each consumer may belong to a consumer group. The fundamental difference between a MOM and Kafka is that the clients will never receive message automatically. They must explicitly ask for a message when they are ready to handle it.
    - Topics – logical collection of messages. Data sent by producers are stored in topics. Consumers subscribe to a specific topic that they are interested in.

- Partition – Each topic is split into one or more partitions. They are like shards and Kafka may use the message key to automatically group similar messages into partition. This scheme enables Kafka to dynamically scale the messaging infrastructure. Partitions are redundantly distributed across the Kafka cluster. Messages are written to one partition but copied to at least two more partitions maintained on different brokers within the cluster.
- Consumer groups – consumers belong to at least one consumer group, which is typically associated with a topic. Each consumer within the group is mapped to one or more partitions of the topic. Kafka will guarantee that a message is only read by a single consumer in the group. Each consumer will read from a partition while tracking the offset. If a consumer that belongs to a specific consumer group goes offline, Kafka can assign the partition to an existing consumer. Similarly, when a new consumer joins the group, it balances the association of partitions with the available consumers.

It is possible for multiple consumer groups to subscribe to the same topic. For example, in the IoT use case, a consumer group might receive messages for real-time processing through an Apache Storm cluster. A different consumer group may also receive messages from the same topic for storing them in HBase for batch processing.

The concept of partitions and consumer groups allows horizontal scalability of the system.

- Broker – Each Kafka instance belonging to a cluster is called a broker. Its primary responsibility is to receive messages from producers, assigning offsets, and finally committing the messages to the disk. Based on the underlying hardware, each broker can easily handle thousands of partitions and millions of messages per second.

  The partitions in a topic may be distributed across multiple brokers. This redundancy ensures the high availability of messages.

- Cluster – A collection of Kafka broker forms the cluster. One of the brokers in the cluster is designated as a controller, which is responsible for handling the administrative operations as well as assigning the partitions to other brokers. The controller also keeps track of broker failures.
- ZooKeeper – Kafka uses Apache ZooKeeper as the distributed configuration store. It forms the backbone of Kafka cluster that continuously monitors the health of the brokers. When new brokers get added to the cluster, ZooKeeper will start utilizing it by creating topics and partitions on it.

## Appendix B. Docker Swarm (mode) [36]

Docker Swarm is Docker's native clustering technology. It works very well with the Docker command line tools like docker and docker-machine, and provides the basic ability to deploy a Docker container to a collection of machines running the Docker Engine. Docker Swarm does differ in scope, however, from what we saw when reviewing Amazon ECS.

Amazon ECS leverages its own technology stack to run Docker containers. This includes EC2 instances to host the virtual machines, auto-scaling to scale those virtual machines up and down, Elastic Load Balancers (ELB) to distribute load to your Docker containers, and more.

Docker Swarm, on the other hand, is only a clustering technology: you register the servers that can run Docker Containers with Swarm and Swarm will deploy containers to those machines. It is your responsibility to start and stop machines, register and deregister machines with Swarm, and register and deregister your containers with your own load balancing solution.

While Amazon ECS is probably the preferred way of clustering Docker in AWS, Docker Swarm does have the ability to run anywhere, including outside of Amazon. Many organizations are embracing the cloud slowly and many organizations are running in a hybrid cloud environment, in which some applications or additional instances of applications are running in a public cloud while the remaining applications are running in a local data center.

Regardless of whether you have embraced a cloud platform, are running a hybrid cloud, or running solely in your own data center, Swarm will enable to you to take advantage of Docker in any environment.

**So how does Docker Swarm work?**

Docker Swarm is implemented using two different types of components:

A manager container, which runs on a virtual machine, manages the environment, deploys containers to the various agents, and reports the container status and deployment information for the cluster; it is your primary interface into Docker Swarm

Agents are containers running on virtual machines that register themselves with the manager and run the actual Docker containers.

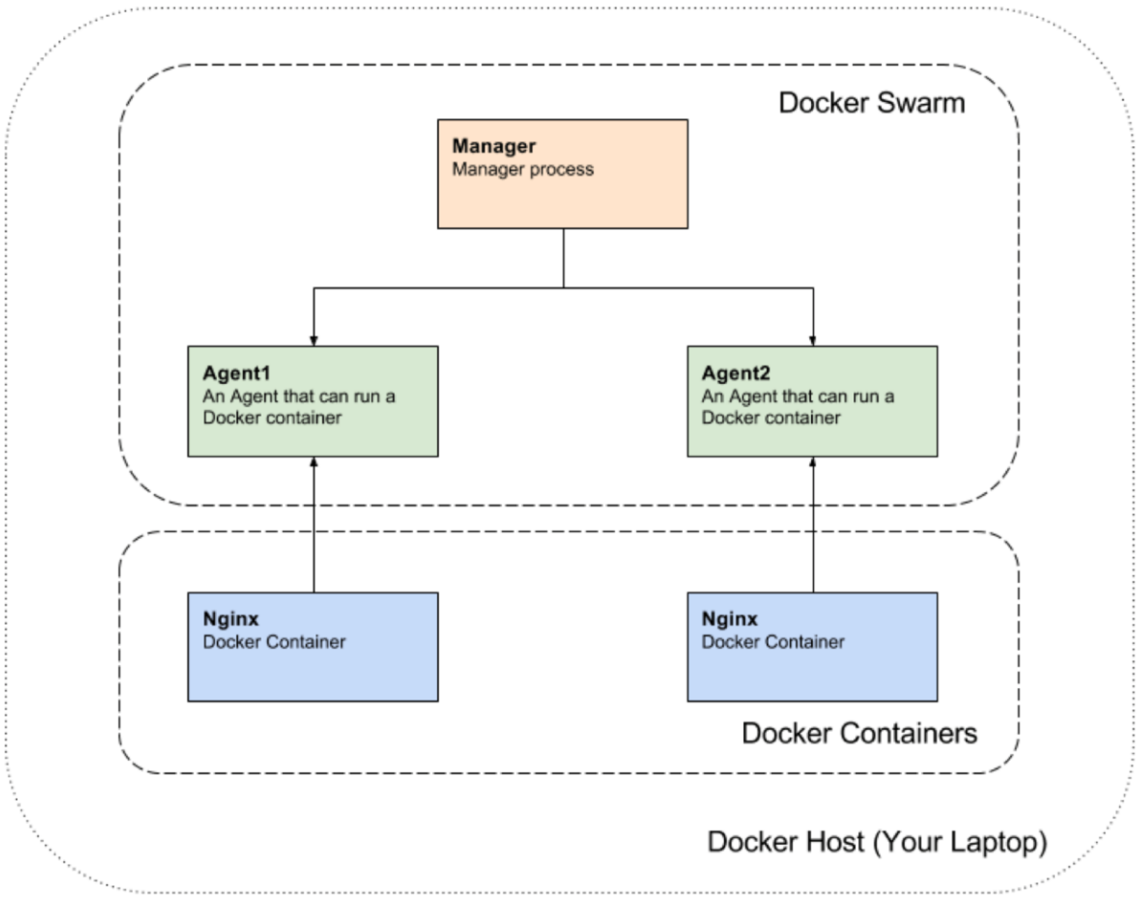

Docker Swarm
Manager
Manager process
Agent1
An Agent that can run a Docker container
Agent2
An Agent that can run a Docker container
Nginx
Docker Container
Nginx
Docker Container
Docker Containers
Docker Host (Your Laptop)

**Figure 36 - Docker Swarm Architecture**

In this example, we have a Docker Swarm Manager that is managing two agents (Agent1 and Agent2). Those two agents are running two instances of an Nginx Container. Both the manager and agents are “docker machines” that contain the Docker Engine and are capable of running Docker containers. As we’ll see in the next section, Docker machines are very similar to Docker containers themselves, with the exception that they are started using the docker-machine command instead of the docker command.

The example in the next section will demonstrate how to setup a Docker Swarm cluster on your local machine (hence the reference to “Your Laptop” as the Docker Host in figure 1). In production, both the manager as well as the agents will run directly on their own virtual machines.

## Appendix C. Docker Flow Proxy [37] [38]

The docker flow proxy uses HAProxy as a proxy and adds custom logic that allows on-demand reconfiguration of the proxy. It provides an easy way to reconfigure proxy every time a new service is deployed, or when a service is scaled.

Docker flow proxy can be configured through docker environment variables and/or by creating a new image based on vfarcic/docker-flow-proxy.

## Appendix D. Prometheus [39]

Prometheus is an open-source systems monitoring and alerting toolkit. It was originally built by SoundCloud.

Following are its features:

- A multi-dimensional data model (time series identified by metric name and key/value pairs)
- A flexible query language to leverage this dimensionality
- No reliance on distributed storage; single server nodes are autonomous
- Time series collection happens via a pull model over HTTP
- Pushing time series is supported via an intermediary gateway
- Targets are discovered via service discovery or static configuration
- Multiple modes of graphing and dashboarding support

Prometheus ecosystem consists of the following components (some of which are optional):

- The main Prometheus server which scrapes and stores time series data
- Client libraries for instrumenting application code
- A push gateway for supporting short-lived jobs
- Special-purpose exporters (for HAProxy, statsD, graphite etc)
- An alert manager
- various support tools

Following diagram shows the architecture:

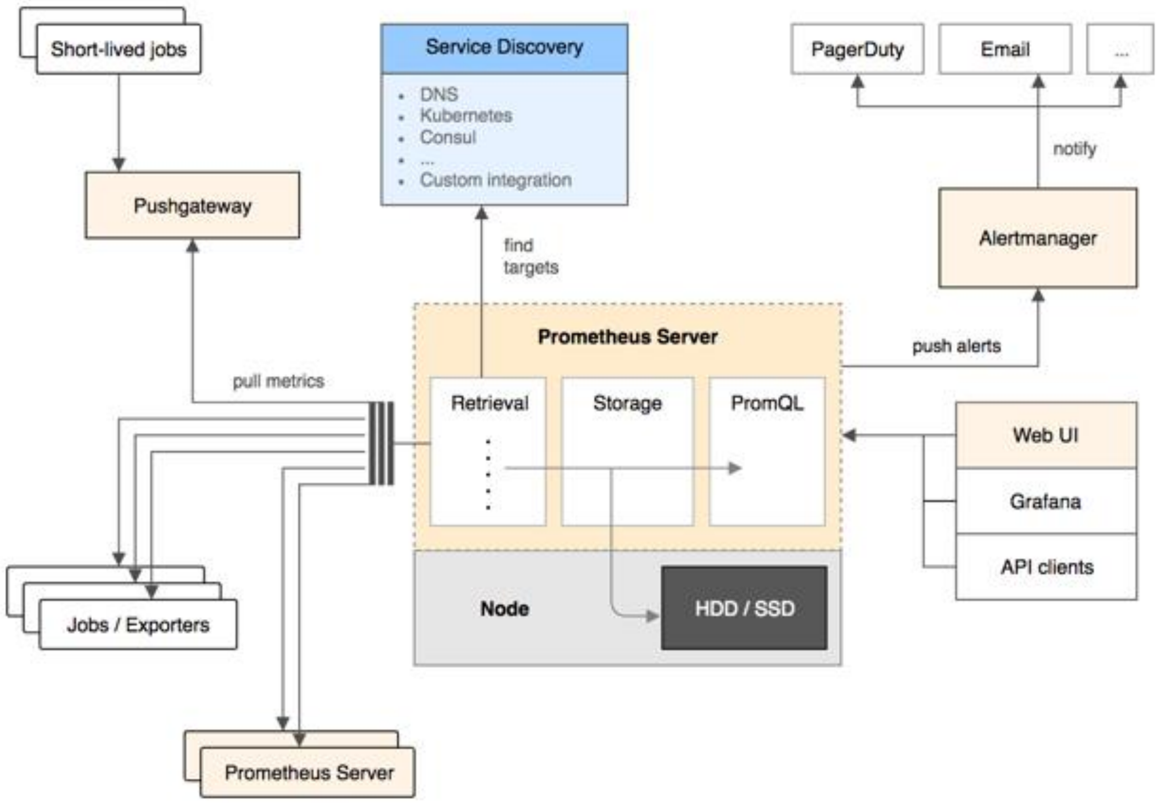


**Figure 37 - Prometheus Architecture**

Prometheus scrapes the metrics from jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped samples locally and runs rules over this data to either record new time series from existing data or generate alerts. Grafana or API consumers can be used to visualize the collected data.

## Appendix E. cAdvisor [40]

cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. This data is exported by container and machine-wide.

cAdvisor has native support for Docker containers and should support just about any other container type out of the box.

## Appendix F. Grafana

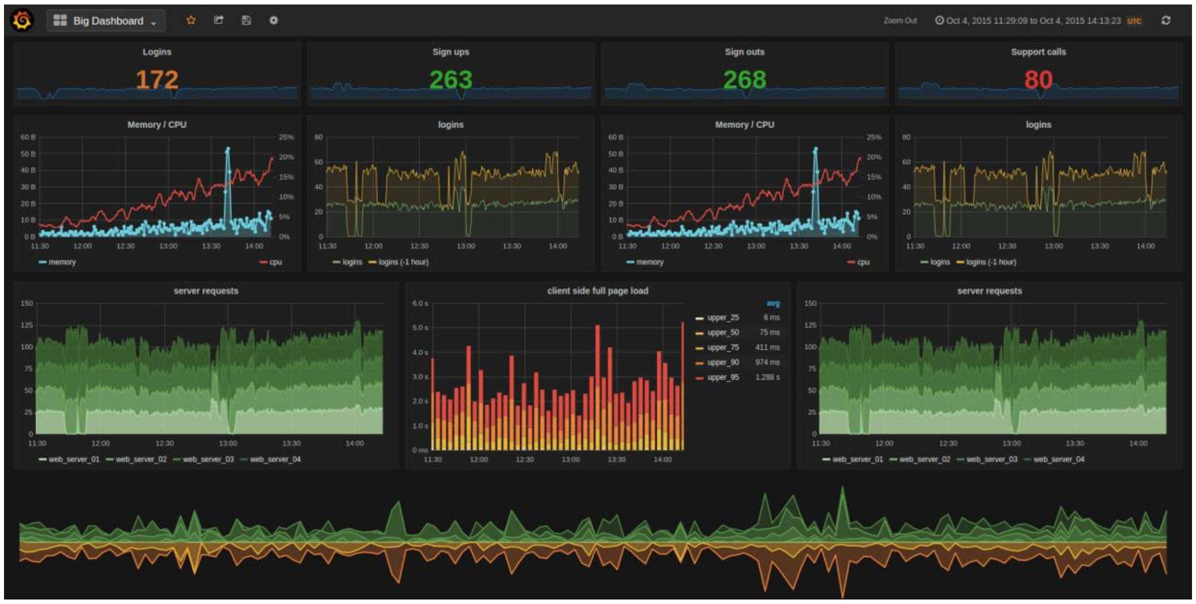


**Figure 38 - Grafana dashboard**

Grafana is an open source, feature rich metrics dashboard and graph editor for Graphite, Elasticsearch, Open TSDB and Prometheus.

Its features include:

- Graphing
    i. Fast rendering, even over large timespans
    ii. Click and drag to zoom
    iii. Multiple y-axis, log scales
    iv. Bars, lines and points
    v. Smart Y-axis formatting
    vi. Legend values and formatting options
    vii. Grid thresholds, axis labels

viii. Any panel can be rendered to PNG

- Dashboards

  i. Create, edit, save and search dashboards

  ii. Change column spans and row heights

  iii. Templating

  iv. Scripted dashboards

  v. Dashboard playlists

  vi. Time range controls

  vii. Share snapshots publicly

- Prometheus

  i. Feature rich query editor UI

- Alerting

  i. Define alert rules using graphs and query conditions

  ii. Schedule and evaluate alert rules, send notifications to Slack, etc.

## Appendix G. Elastic Stack [41]

The ELK stack consists of Elasticsearch, Logstash and Kibana. Elasticsearch along with Logstash and Kibana, provides a powerful platform for indexing, searching and analyzing data.

Data constantly flows into your systems, but it can quickly grow to be fat and stale. As your data set grows larger, your analytics will slow up, resulting in sluggish insights. And this is likely to be a serious business problem. So, the BIG question for your big data is: how can you maintain valuable business insights?

- Elasticsearch is a log search tool with following benefits:
    - Real-time data and real-time analytics
    - Scalable, high-availability and multi-tenant
    - Full text search
    - Document oriented – data stored as JSON documents
- Logstash is a tool for log data intake, processing and output. It is a pipeline for event processing, it takes precious little tie to choose the inputs, configures the filters and extract the relevant, high-value data from logs. Then integrate with Elasticsearch to persist the filtered data and super-fast queries against mountains of data.
- Kibana is for visualizing log data in dashboard. It is used to visualize logs and other time stamped data.