Analysis of queuing services in a distributed architecture

|  |
| --- |
| Project Workbook-2 |
|  |

|  |
| --- |
| By |
| Atish Maitreya 013708292  Daniel Sampreeth Reddy Eadara 013702988  Sanjay Nag Bangalore Ravishankar 013591851  Viraj Upadhyay 012490959 |
| Nov 7th, 2019 |
| **Advisor:** Dr. Gokay Saldamli |

**Table of Contents**

**Chapter 1. Literature Search, State of the Art 3**

Literature Search 3

State-of-the-Art Summary 6

References 7

**Chapter 2. Project Justification 11**

**Chapter 3. Identify Baseline Approaches 12**

**Chapter 4. Dependencies and Deliverables 14**

Dependencies 14

Deliverables 14

**Chapter 5. Project Architecture 15**

**Chapter 6. Evaluation Methodology 17**

**Chapter 7. System Design/ Methodology 19**

**Chapter 8. Implementation Plan and Progress 29**

**Chapter 9. Project Schedule 31**

# Literature Search, State of the Art

## Literature Search

Distributed architecture is a widely used software architecture. The other major form of architecture is Monolithic architecture. A monolithic architecture is a tightly coupled architecture that bundles all functionalities in a single application. This type of architecture is usually great, to begin with when the code-base is fairly small, the team is still in uncharted territory and you cannot afford to maintain and monitor multiple code-bases [2]. Microservice architecture is a type of distributed architecture that we will be following. In this type of architecture, a service is an isolated functional unit that deals with a well defined primitive task [1]. These services communicate using language-agnostic API’s [1]. The major advantages of using microservice architecture are scalability and reusability. These services have to communicate with others sometimes, a popular method to accomplish this is using RESTful methods to communicate over HTTP. This communication method can get sluggish if the scale of the application is big. Message queues are a solution to this problem. They enable this by allowing applications to send messages asynchronously to one another [3]. These queues also have the capability of storing messages when the receiver is busy or not available [3].

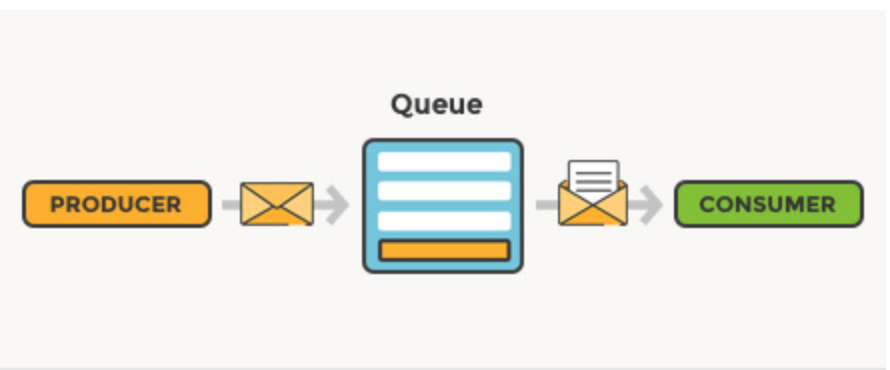


Fig 1.1 Message Queue Architecture [3]

There are many message queues and all of them provide different results. These systems perform differently in terms of reliability, scaling, and partitioning [4]. There are many implementations of queues like RabbitMQ, Kafa and Amazon SQS. RabbitMQ is open-sourced and it has a good and mature community. Kafa has a complex architecture and it is governed by a “zookeeper” [5]. Amazon SQS is a managed queueing service provided by AWS. “SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery. SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.” [6].

One of the properties to judge these implementations is throughput. Kafka is known for its high throughput. Amazon SQS limits its users on the message rate. Compared to both of these, rabbitmq is easily configurable but the management of the cluster can be complicated.

The problem is choosing a provider for a specific application. “The answer is to research that compares the processing speed for sending and receiving, the memory load, the platforms that brokers can manage, etc”. [7]. Also, we plan to get better results by tweaking the configurations and adjusting the architecture could provide significant performance improvements.

## State-of-the-Art Summary

* Our research proposes a comparison framework for quantitative analysis of messaging services. This framework would recommend messaging queues, based on the required characteristics of the application that is being developed. These characteristics can be Scalability, Availability, Push Queue requirements, Pull Queue requirements, Fault tolerance, Partition recovery, etc.
* State-of-the-art messaging systems such as Apache Kafka, Amazon SQS and RabbitMQ are used in this research. The above three eclectic sets of messaging services exhibit a wide gamut of characteristics, which can be recommended for most requirements.
* A modern front-end framework like ReactJS is used to create client web pages. Using this, modular, fast, lightweight and user-friendly front-end user interface will be created. The state management on the client-side will be using Redux.
* Our research project incorporates replication and sharding for our No-SQL datastores. This would ensure high scalability, high availability, eventual consistency, and partition tolerance. Most of the state-of-the-art web applications provide the above characteristics to their users.
* This research project uses microservices architecture. Microservices ensures scalability as one of its most salient features. This would aid a containerized approach to the applications. Containerizing the application would help in easy management and fault tolerance of the application.

## References

1. D. S. Linthicum, "Practical Use of Microservices in Moving Workloads to the Cloud," in IEEE Cloud Computing, vol. 3, no. 5, pp. 6-9, Sept.-Oct. 2016.

doi: 10.1109/MCC.2016.114

keywords: {cloud computing;service-oriented architecture;microservices;moving workloads;cloud;enterprises;service-oriented applications;monolithic application;Cloud computing;Information storage;Computer architecture;Computer security;Service-oriented architecture;Complexity theory;Information management;cloud computing;microservices;service-oriented architecture;containers;Cloud Tidbits},

URL: <http://ieeexplore.ieee.org.libaccess.sjlibrary.org/stamp/stamp.jsp?tp=&arnumber=7742277&isnumber=7742214>

1. A. Koschel, I. Astrova and J. Dötterl, "Making the move to microservice architecture," 2017 International Conference on Information Society (i-Society), Dublin, 2017, pp. 74-79.

doi: 10.23919/i-Society.2017.8354675

keywords: {project management;service-oriented architecture;software management;microservice architecture;software projects;on-demand scalability;Computer architecture;Companies;Software;Servers;Web pages;Databases;monolithycal architecture;microservice architecture;large software projects},

URL: <http://ieeexplore.ieee.org.libaccess.sjlibrary.org/stamp/stamp.jsp?tp=&arnumber=8354675&isnumber=8354650>

1. Lovisa Johansson. What is message queuing? (2019,Sep 25)

Retrieved Sep 29,2019

from <https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html#targetText=A%20message%20queue%20is%20a,with%20some%20headers%20on%20top>.

1. John, Vineet and Xia Liu. “A Survey of Distributed Message Broker Queues.” ArXiv abs/1704.00411 (2017): n. pag.

1. Antoine Duprat. How to choose a message queue. (2018, Feb 9)

Retrieved Sep 29,2019

<https://medium.com/linagora-engineering/how-to-choose-a-message-queue-247dde46e66c>

1. Amazon Simple Queuing Service.

Retrieved Sep 29,2019

<https://aws.amazon.com/sqs/>

1. V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), Craiova, 2015, pp. 132-137.

doi: 10.1109/RoEduNet.2015.7311982

keywords: {message passing;middleware;performance analysis;network connected software applications;data transfer;clients;servers;data storage;asynchronous communication;message oriented middleware;MOM;message queues;message brokers;RabbitMQ brokers;ActiveMQ brokers;message sending;message receiving;Decision support systems;message queues;Message Oriented Middleware;broker;RabbitMQ;ActiveMQ;compare},

URL: <http://ieeexplore.ieee.org.libaccess.sjlibrary.org/stamp/stamp.jsp?tp=&arnumber=7311982&isnumber=7311815>

1. Roman Wiatr, Renata Słota, Jacek Kitowski,

Optimising Kafka for stream processing in latency sensitive systems,

Procedia Computer Science,

Volume 136,

2018,

Pages 99-108,

ISSN 1877-0509,

https://doi.org/10.1016/j.procs.2018.08.242.

(http://www.sciencedirect.com/science/article/pii/S1877050918315473)

# Project Justification

Distributed systems are a group of systems working concurrently with each other. Their principle context involves independent systems interlinked and enabled with the ability of working as an integrated unit. Even if a single system fails, the other systems still have the capability to work autonomously. They collaborate with each other by communicating and this is where the concept of message queues arrives. A message queue contributes for transmitting a sequence of data among multiple systems. They also offer temporary storage of data and the data transfer is asynchronous.

Lately, multiple message queues like AmazonSQS, RabbitMQ and Kafka have come into existence, each with their own exclusive capabilities and properties. They vary in scalability, cost, throughput, and reliability. Performance and efficiency are also to be looked out for in a message queue along with a choice between pull and push types. A push message queue forwards the data from the dealer directly, while the pull message queue depends on the subscriber requirement.

Currently, our aim is to build an application that measures and evaluates the prominent features of the message queues. These will be tested under various workload architectures and distinct architectures. The results obtained will prove as the basis in deciding the optimal message queue for meeting one’s standards and requirements.

# Identify Baseline Approaches

Message queueing systems store messages that are to be passed around different systems. Some majorly used messaging systems are Kafka, SQS, and RabbitMQ. Systems like Kafka and Kinesis can also be used for streaming data.

SQS is a fully managed service by amazon. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available[6]. Kafka is used for building real-time streaming data pipelines. While RabbitMQ is an open-source queuing option.

We are planning to create a project that leverages the above-mentioned technologies compares the performance to form a framework for the quantitative comparison of the queues. For this, we will create an application to gauge the capabilities of the aforementioned queues. This application will be a distributed web app which will be divided into microservices. These microservices will be containerized using docker as doing this will make deployment and scaling easier. Hence, this architecture will be highly scalable. We also plan to orchestrate the deployment of the containers using Kubernetes. Our databases will be NoSql clusters which will have replication and sharding for fault tolerance and high performance. We also plan to use replication in our queuing service to increase the efficiency and reliability of the queues. These systems will be benchmarked with different queues in different scenarios and configurations. Along with this we also plan to improve the performance of these queues by studying them and changing the configurations and/or infrastructure. We also plan to use tracing frameworks embedded in our code to get further insights into the working of our system.

# Dependencies and Deliverables

## Dependencies

* Limitations in computational resources.
* Identifying assets for cloud infrastructure.
* Tweaking parameters of messaging queues according to the requirement. This requires architectural changes in these messaging queues, consequently forming a dependency on product development.
* Storing and archiving infrastructure logs. Usage of public cloud services like Amazon Cloud Trail will quicken our development cycle instead of building our own infrastructure for logs.

## Deliverables

* An application with different configurations to test run the message queues.
* A detailed report on benchmark results of comparison between message queues.
* Statistical analysis of message queues in multiple architectures.
* Statistical analysis of message queues in various workloads.

# Project Architecture

Our research aims at providing a quantitative analysis of messaging services and suggest messaging queues based on the application requirement characteristics. Figure 5.1 depicts the architecture of the project.

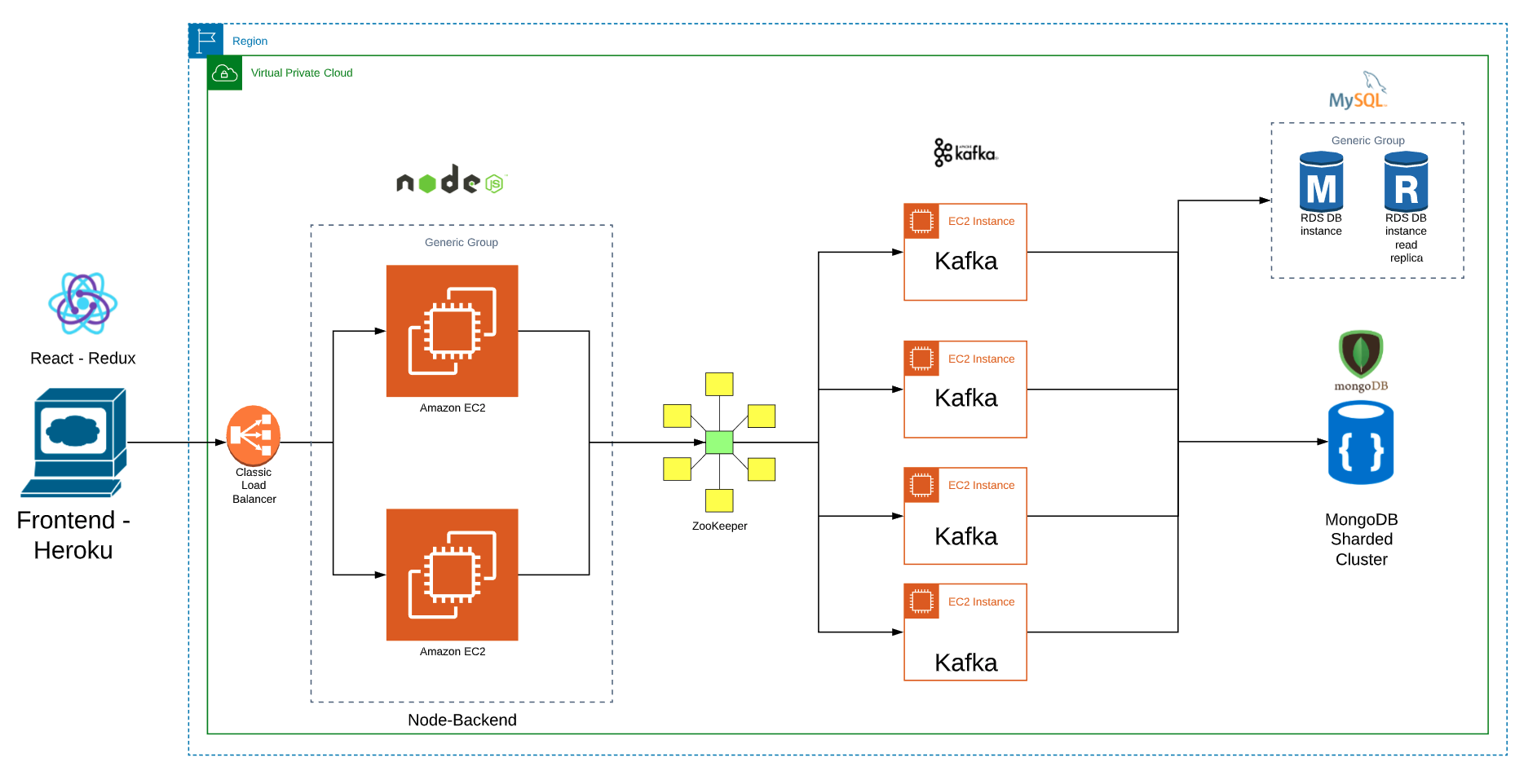


Fig 5.1 Project Architecture

**Frontend -** Front end uses React-Redux framework for all the web pages. In addition to that technologies like HTML, CSS, JavaScript, and JQuery are used in developing interactive web pages. The front end application will be hosted on Heroku.

**Middleware -** NodeJS servers will be acting like Middleware. These contain application login APIs. This will be containerized and hosted on a Virtual Private Cloud (VPC) in Amazon Web Services (AWS). We have multiple instances of these containers and to provide scalability, we have a load balancer. This load balancer will auto-scale the middleware containers based on the number of requests. When there is a request from the frontend, this hits a load balancer. This request from load balancer is forwarded to one of the middleware containers.

**Messaging Queue -** Messaging queues can be either Apache Kafka, Amazon SQS or RabbitMQ. This selection of the messaging queue is based on the quantitative analysis performed above in this article. In this figure, Apache Kafka is depicted as the messaging queue. The processed request from the middleware containers will be directed to a messaging queue. This request would then be forwarded to the Datasource. These Apache Kafka servers are hosted on AWS.

**Database -** The requests from the messaging queue are consumed by the Datasource. We have multiple types of data sources. We have used MongoDB for Application Data. These data will be used by the application logic and is domain-specific. We will provide replication and sharding capabilities to these to provide high scalability, availability, throughput and partition tolerance. SQL database is also used to decouple user login data from the application data, which ensure security to user login data. This will also prevent any accidental changes to user login data from the application APIs.

# Evaluation Methodology

We follow a test-driven development approach. The following tests will be conducted for evaluation purposes.

**Validation testing -** We validate our backend REST APIs using the Mocha-Chai test framework. Using this framework, we select a Rest API and provide a predetermined input for that particular API. We compare the expected output with the actual output to validate that actual output is, in fact, coherent with what is expected. We ensure there are assertions for the actual output as any change of the output is easily found out by the particular test case failure.

**Scalability Testing** - For scalability testing, we intend to test with huge predetermined user requests, at a single point in time. The capability of message queues will be tested while trying to handle this huge load. The number of user requests will be increased constantly. First, the number of message queues will be kept constant and the number of user requests will be tested under a constantly increasing load. Secondly, the number of user requests from multiple users will be tested under constantly increasing message queue nodes. The endurance of each message queue based on the number of user requests will be noted for further comparison. For the above-mentioned load testing, we use the Apache JMeter test framework for automated testing and use POSTMAN for manual.

**Availability Testing** - Availability testing is primarily done as MongoDB while under partition is a CP system according to the CAP (Consistency Availability Partition tolerance) theorem. Since availability is integral to all standard applications, we would like to follow suit. We test the availability of data through the application. We purposely kill servers and check how our applications work. We employ sharding methodologies in MongoDB bake availability into our application design. We will test these by using POSTMAN and Mocha-Chai test framework.

# **System Design / Methodology**

**Block Diagram**

Below block diagram depicts one of the queues - Apache Kafka and its implementation with our application. Here a Node Backend Service pushes data into a Kafka topic. These messages are then mapped to appropriate Kafka topics and the necessary database request, based upon the user action gets executed. Response to these database queries is returned to Kafka which in turn transfers the resulting data Node Backend service subsequently initiating an appropriate action on the user interface.

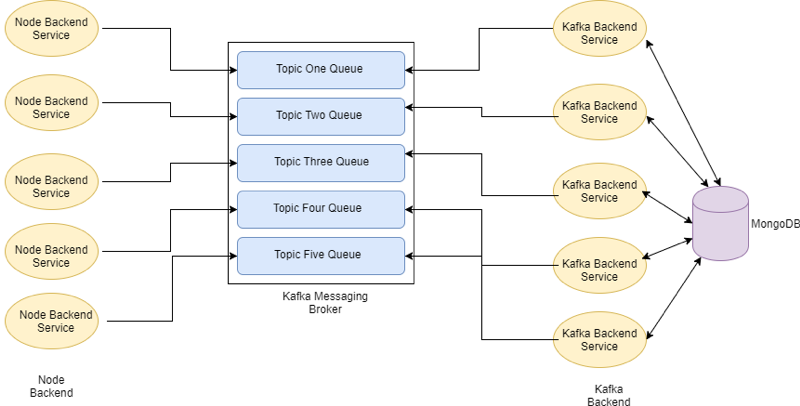


Fig 7.1 Project Block Diagram

**Sequence diagram**

The sequence diagram below depicts the SignUp and Login activities provided by the application. This also shows the different phases of request/response and also briefly indicates the functionalities and modifications of data across each phase.

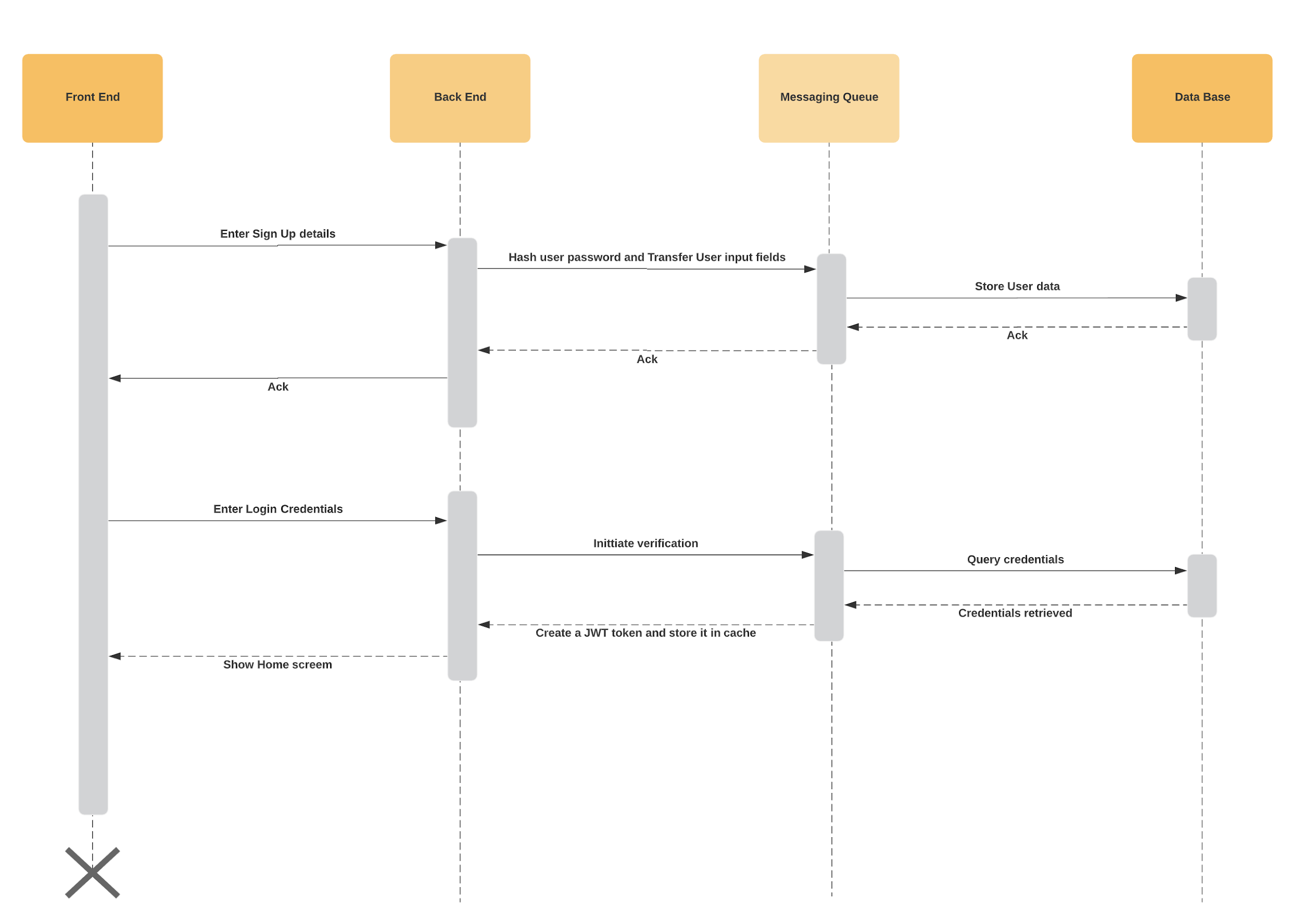


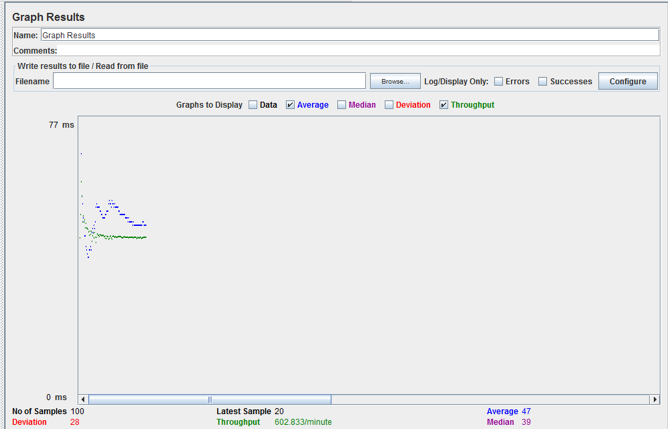
Fig 7.2 Project Sequence Diagram

**Baseline evaluation**

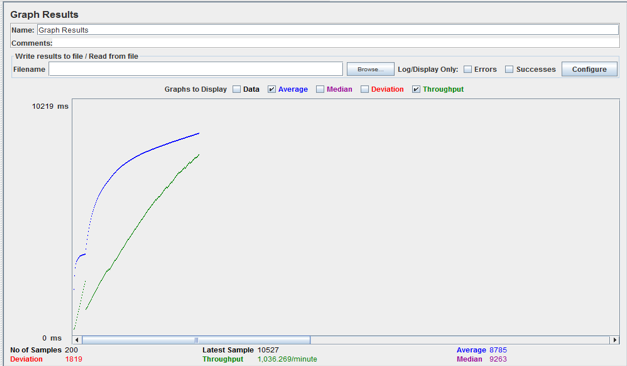
Apache Kafka is one of our baselines. We have implemented Signup and Login using Apache Kafka queues. As shown in the sequence diagram in Fig 7.2, signup and login phases contain many transformations/modifications as it passes from the frontend to the database and back. We have evaluated the performance of this by using the JMeter test framework. We have tested both services with 100, 200, 300, 400 and 500 concurrent users.

**SignUp**

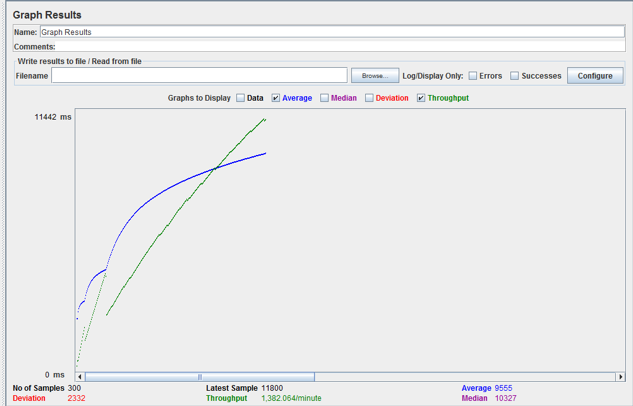
**100 concurrent users**

****

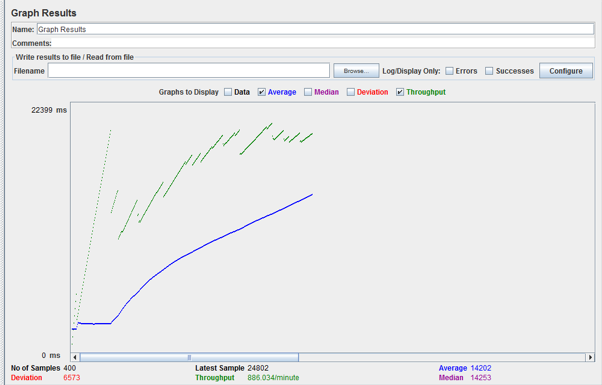
**200 concurrent users**

****

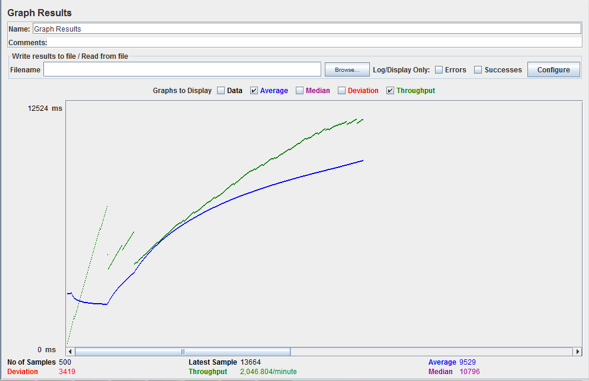
**300 concurrent users**

****

**400 concurrent users**

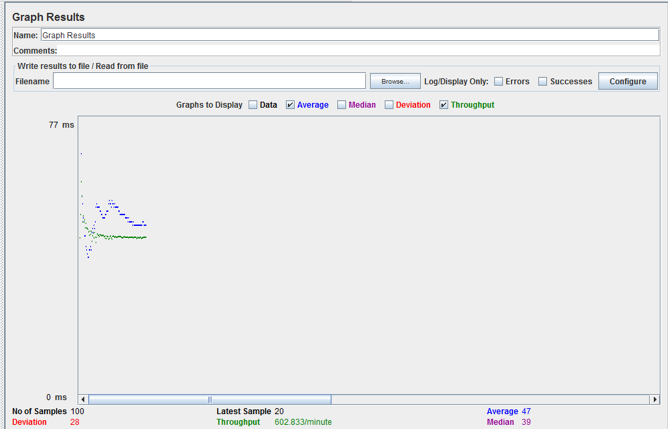
****

**500 concurrent users**

****

**Login**

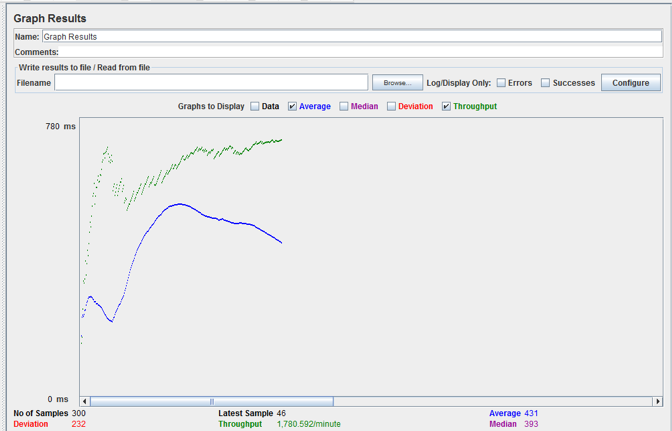
**100 concurrent users**

****

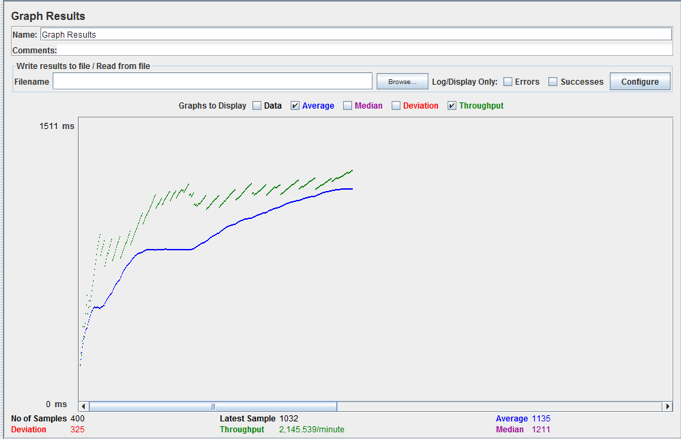
**200 concurrent users**

****

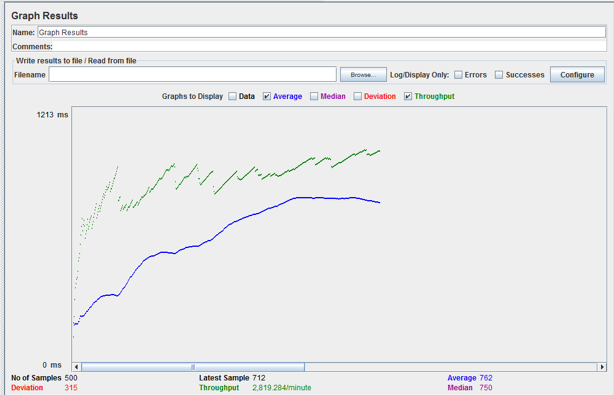
**300 concurrent users**

****

**400 concurrent users**

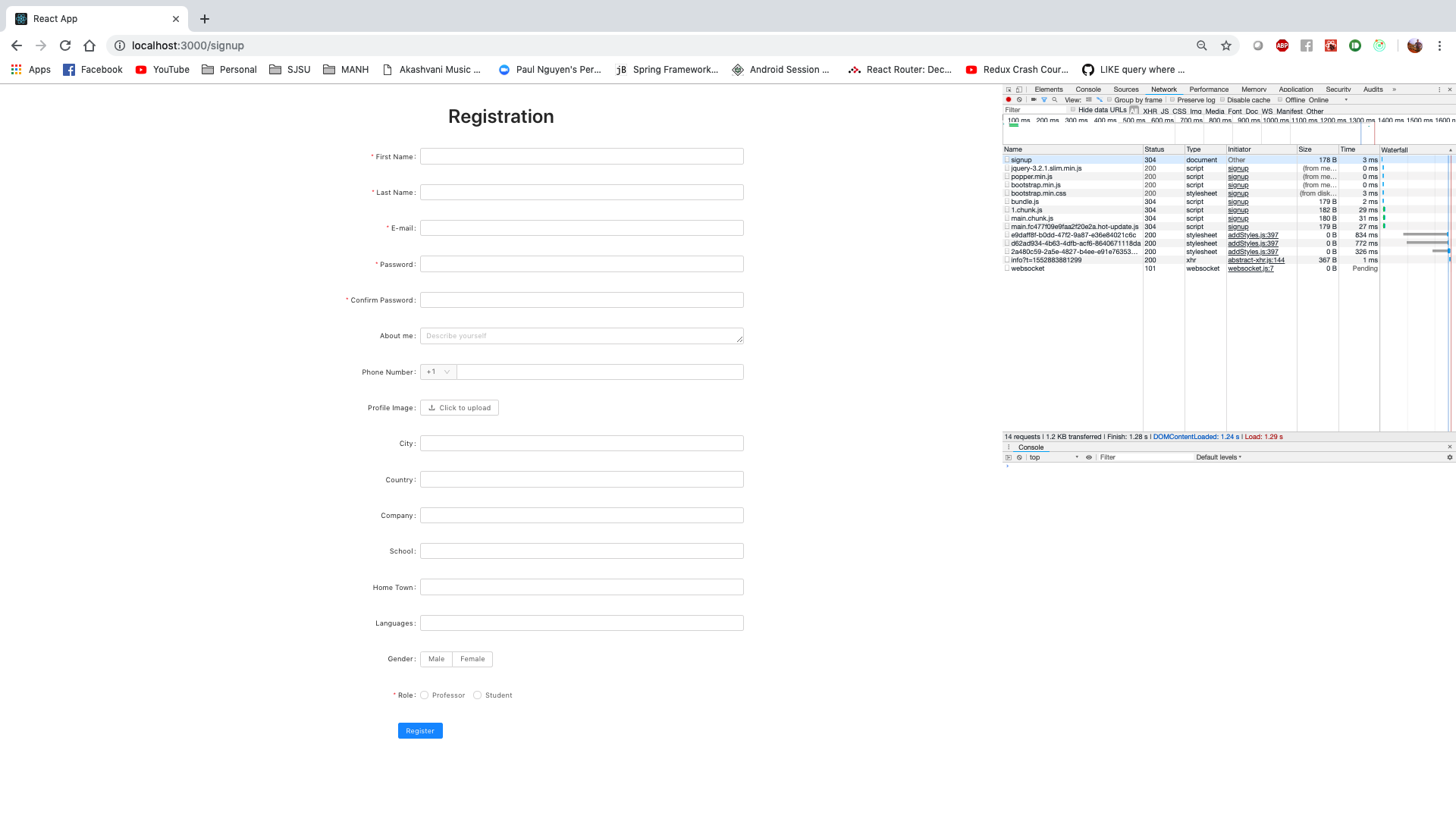
****

**500 concurrent users**

****

**User Interface Mockups**

The following UI depicts the rudimentary user interface. This includes a signup page for different personas. and login page for all these respective personas.



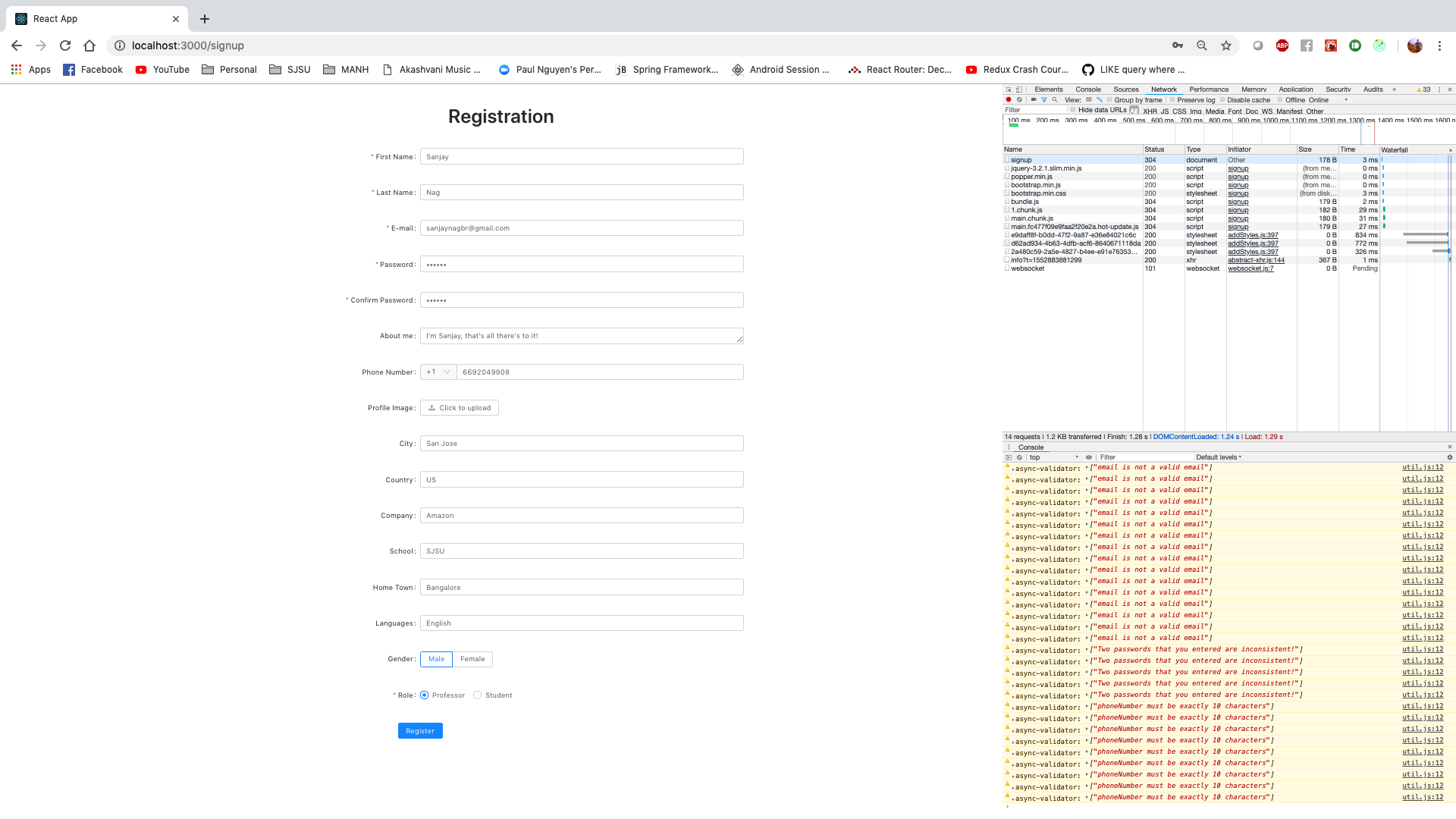


Fig 7.3 Registration Pages

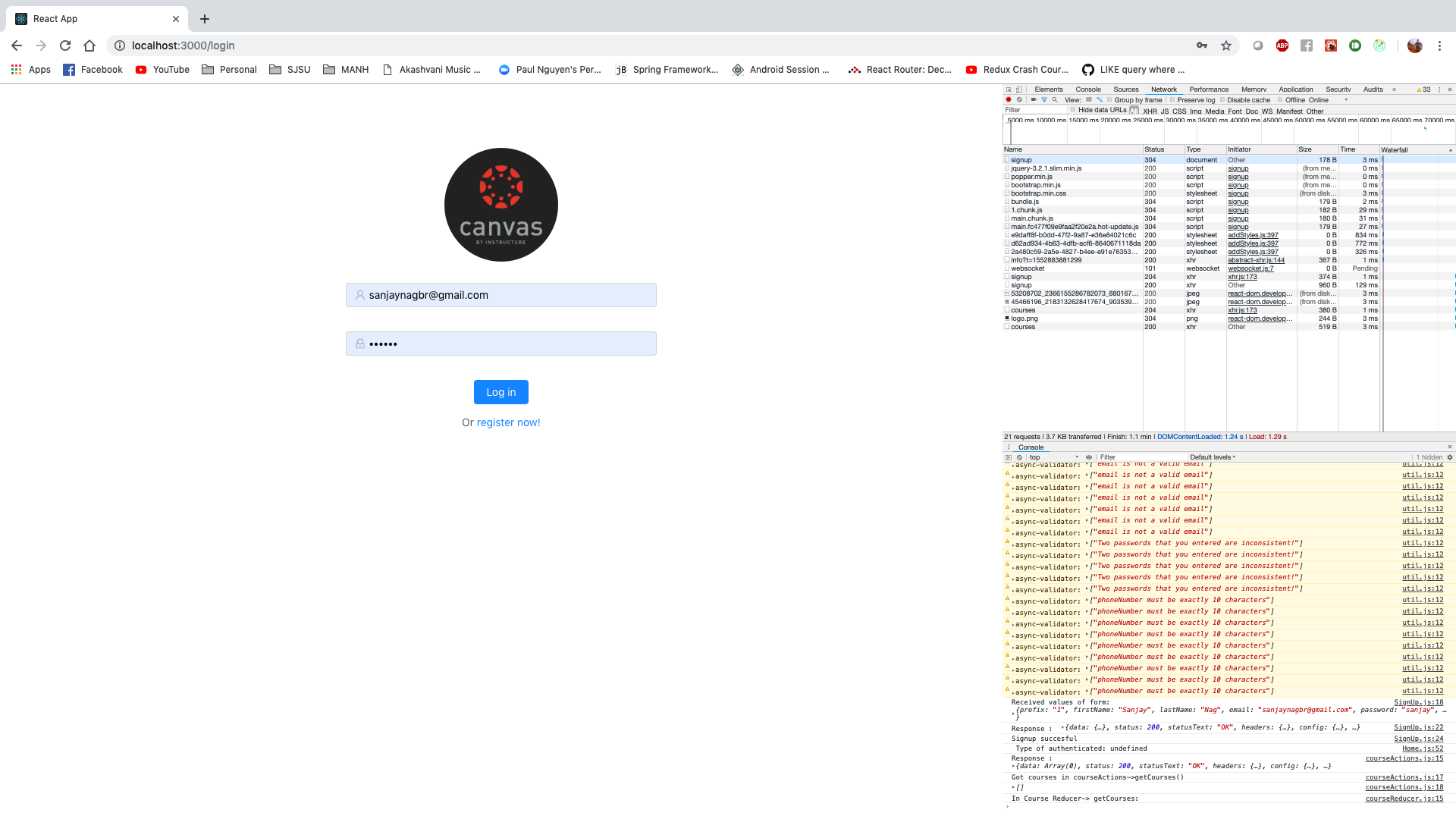


Fig 7.4 Login Page

# Implementation Plan and Progress

The project implementation will be achieved through various phases. Multiple steps are required for each phase. The first phase is deciding the technologies to be used for the purpose of this project. Step one is to choose an application that can be used to test the performance of the message queues. Step two is to choose among the available message queues for the purpose of this project. The last step is finalizing the scope of this project.

The second phase is the development phase where the aforementioned technologies are combined to create a working application. It includes several steps: first being the decision making of choosing the right technology stack for the application, the second being the actual development of this application, and lastly, plugging in the different message queues to this application.

The final phase includes hosting and benchmarking this application. Various cloud platforms are used to host the application and benchmark it using the right tools. The results obtained during this test are the conclusion of this project. Regular documentation, logbooks and reporting are also carried out across the entire timeline. To summarize, the following is a summary of the project plan in sequence.

* Research and literature review.
* Phase one implementation.
* Finalizing the technologies and scope of the project.
* Workbook and logbooks.
* Phase two implementation.
* Finishing up the development.
* Phase three implementation.
* Benchmarking results and conclusion
* Final review and report

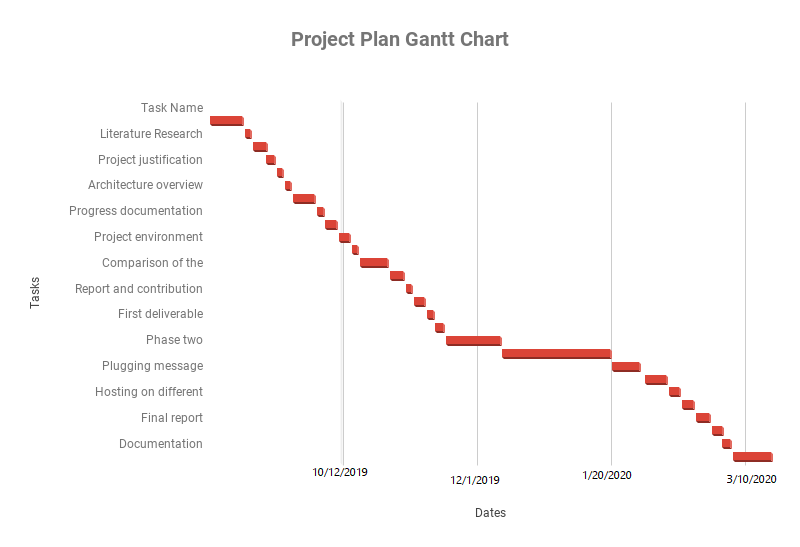
# Project Schedule

V: Viraj Upadhyay, A: Atish Maitreya, SR: Sampreeth Reddy, SN: Sanjay Nag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task Name** | **Begin Date** | **End Date** | **Assigned to** | **Status** |
| Project Abstract | 8/24/2019 | 9/5/2019 | All | Complete |
| Literature Research | 9/6/2019 | 9/8/2019 | A | Complete |
| State of the art Research | 9/9/2019 | 9/14/2019 | SN | Complete |
| Project justification | 9/14/2019 | 9/17/2019 | SR | Complete |
| Project planning | 9/18/2019 | 9/20/2019 | V | Complete |
| Architecture overview | 9/21/2019 | 9/23/2019 | SN | Complete |
| Workbook 1 | 9/24/2019 | 10/2/2019 | All | Complete |
| Progress documentation | 10/3/2019 | 10/5/2019 | V | Complete |
| Finalizing technologies | 10/6/2019 | 10/10/2019 | All | Complete |
| Project environment analysis | 10/11/2019 | 10/15/2019 | SR | Complete |
| Dependencies analysis | 10/16/2019 | 10/18/2019 | A | Complete |
| Comparison of the message queues | 10/19/2019 | 10/29/2019 | All | Complete |
| Workbook 2 and Logbook | 10/30/2019 | 11/4/2019 | All | Complete |
| Report and contribution | 11/5/2019 | 11/7/2019 | SN, V | In progress |
| Phase one implementation | 11/8/2019 | 11/12/2019 | SN | In progress |
| First deliverable | 11/13/2019 | 11/15/2019 | A | Planned |
| Report on the first deliverable | 11/16/2019 | 11/19/2019 | A | Planned |
| Phase two implementation | 11/20/2019 | 12/10/2019 | All | Planned |
| Development of application | 12/11/2019 | 1/20/2020 | TBA | Planned |
| Plugging message queues | 1/21/2020 | 2/1/2020 | TBA | Planned |
| Phase three implementation | 2/2/2020 | 2/10/2020 | TBA | Planned |
| Hosting on different platforms | 2/11/2020 | 2/15/2020 | TBA | Planned |
| Benchmarking | 2/16/2020 | 2/20/2020 | TBA | Planned |
| Final report | 2/21/2020 | 2/26/2020 | TBA | Planned |
| Evaluating the results | 2/27/2020 | 3/1/2020 | TBA | Planned |
| Documentation | 3/2/2020 | 3/5/2020 | TBA | Planned |
| Review | 3/6/2020 | 3/20/2020 | TBA | Planned |

Table 7.1 Project Schedule

## Gantt Chart:



## PERT Chart:

