**Popeyes Management App**

IS 436-03 Structured Systems Analysis and Design Project

Deliverable 4: Data Modeling and Starting Design



Group Members:

Jessica Park

Farah Siddiqui

Betelhem Taddele

Danny Nguyen

Jatin Chauhan

Munam Ghazanfar

Ninh Ngo

Adil Mufti

05/09/2019

Table of Contents

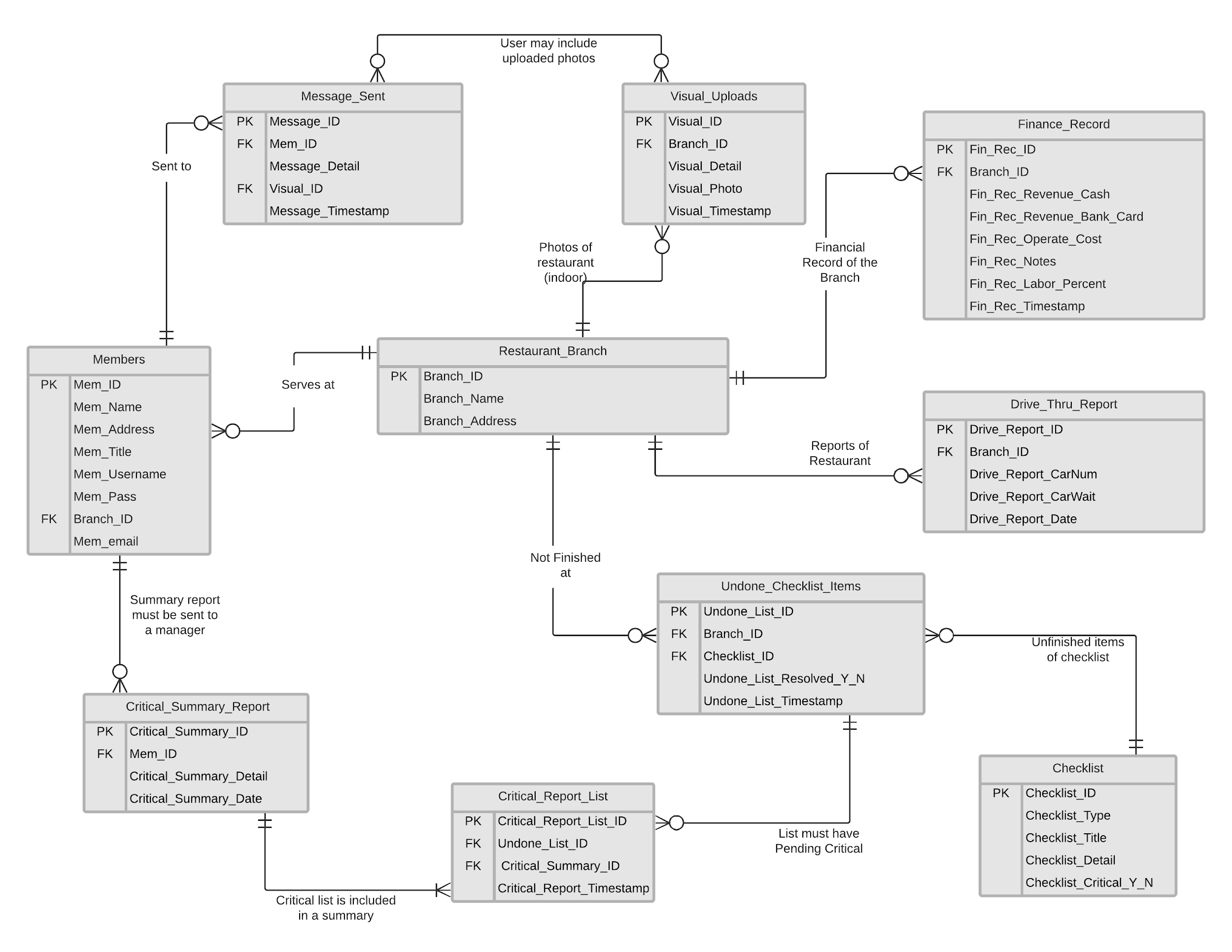
Section 4: Data Modeling Document ........................................................................................................ Figure 4.1 Entity-Relationship Diagram .......................................................................................................2 Entity-Relationship Text Definitions.............................................................................................................3

Alternative Matrix ................................................................................................................................. Figure 4.2 Alternative Matrix ......................................................................................................................4

Figure 4.3 System Architecture Alternative Matrix ....................................................................................6

Figure 4.4 Hardware and Software Specification Custom Built.................................................................. 9

**Entity Relationship Diagram**



|  |  |
| --- | --- |
| **Entities** | **Relationship** |
| Members - Restaurant\_Branch | A Member must have a restaurant’s branch, where he/she works at. |
| Restaurant\_Branch - Finance\_Record | Financial report of restaurant branch will be filled at the end of day. |
| Restaurant\_Branch - Drive\_Thru\_Report | Drive thru report is filed for each restaurant at end of day. |
| Restaurant\_Branch - Visual\_Uploads | Photos of completion is uploaded for every restaurant. |
| Restaurant\_Branch - Undone\_Checklist\_Items | Every unfinished item from checklist in a restaurant is saved so that the owner could see it. |
| Undone\_Checklist\_Items - Checklists | Undone checklist must have item from checklist, that was not finished. |
| Critical\_Summary\_Report -  Members | Critical summary consists every member’s critical task, which was not done. |
| Critical\_Summary\_Report - Critical\_Report\_List | Critical list has multiple items for summary and summary must have at least one list item. |
| Critical\_Report\_List -  Undone\_Checklist\_Items | Each item in critical report list must have the undone item. However, every undone item may not be included in critical list. |
| Message\_Sent - Members | A message is sent to a member and a member can receive multiple messages. |

**Alternative Matrix Descriptions**

Since our application is custom built, the alternative situations we needed to examine were storage. In our research, we looked at 3 different ways to store our application and went about selecting the best option through various metrics such as cost, technical issues, and scalability.

**Alternative 1: Personal Local Server**

* The idea behind using a personal local server comes from discussions during the design process about streamlining the application and allowing the stakeholder to have a more personal approach to the application usage. While the stakeholder is more technically incline, they have no real experience with AWS so opting for that option means a delayed phase after hand-off since they would need to adapt to the AWS environment. Another issue with the personal server is the cost, while the application itself is not compute intensive, there still is reason to purchase a personal separate server as opposed to using a personal computer. The costs with this option is the highest amongst the three and requires some level of tutoring to work with the Microsoft Server Essentials OS, which is another high cost option. The research we did showed a server tower for $1,300 that had, **CPU -** Xeon E3-1220V6 / 3 GHz, **RAM -** 8 GB, **HDD -** 1 TB along with 4 extra HDD space and hot-swappable bays, but this approach would mean we still have to spend another $300 for an OS and more HDD, which should honestly be upgraded to SDD for faster read/write services. The only true positive about this solution is that the server is localized and allows for more personal interaction between the stakeholder.

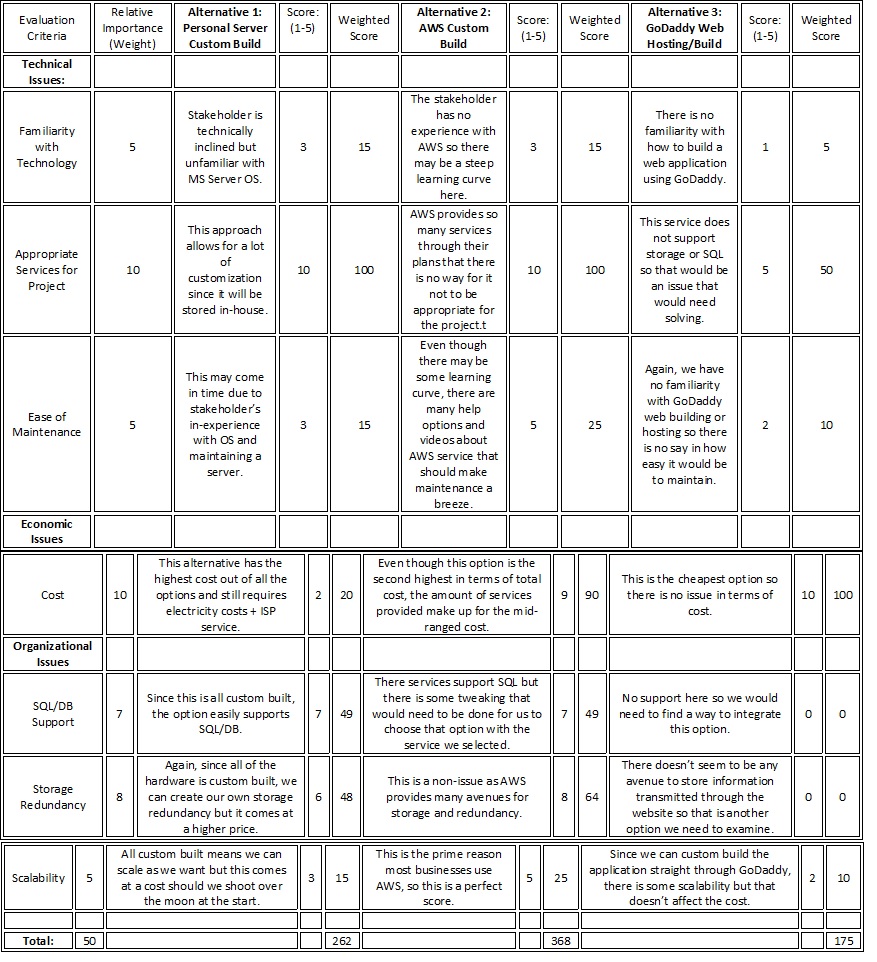
**Alternative 2: AWS Service**

* The option that seems to be the top choice amongst our team. The AWS service is well known and widely supported to various areas of IT business so there is no shortage of troubleshooting options should there be any issues with maintenance for the stakeholder. The other benefit is that the service is entirely scalable so should the features that are initially chosen seem to be too little or too much, the stakeholder has the option to change it at will. The service also comes with storage, SQL support, and a bunch of security/redundancy capabilities that allow for safer storage in the long run. The negatives are the cost, though much cheaper than option 1 but more expensive than option 3 and the learning curve that will come with using AWS for the stakeholder.

**Alternative 3: Aggregate Web Hosting Service (GoDaddy)**

* The third option we came up with is hosting the application through a domain hosting service like GoDaddy. This is a service that no one on the team, nor the stakeholder has experience with. This option means that this application would have to follow a build allowed by GoDaddy, which may or may not be a viable option due to requirements set forth in the beginning of the project. There seems to be no scalable option either with the application or web hosting so that is another issue that fails compared to the other two options. The price point is the highlight of this option though as it is the cheapest of the alternatives. There seems to be no option for some sort of DB option, which is an issue that would need to be looked into before using this service. So while this alternative is the cheapest, it seems to be the least appropriate for our project.

**Alternative Matrix**

****

**System Architecture Alternative Matrix:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirement** | **Server based** | **Thin client-server** | **Thick client-server** |
| **Operational requirement** | | | |
| Technical Requirement | ✔ | ✔ | ✔ |
| System Integration Requirement | ✔ |  | ✔ |
| **Performance Requirement** | | | |
| Speed Requirement |  | ✔ |  |
| Capacity Requirement |  | ✔ | ✔ |
| Availability/ Reliability Requirements | ✔ | ✔ | ✔ |
| **Security Requirements** | | | |
| User Authorization Requirements | ✔ | ✔ |  |
| Access Control Requirements | ✔ | ✔ | ✔ |
| Encryption/Authentication Requirements |  | ✔ | ✔ |
| **Cultural/Political Requirements:** | | | |
| Legal Requirements |  | ✔ |  |
| Popeyes Policy | ✔ | ✔ | ✔ |
| Popeyes Ethic Policy | ✔ | ✔ | ✔ |

**Operational Requirements:**

1. Technical Requirement

1.1. With a thin-client server, the technical Requirement would have an easier time integrating with a cloud/internet platform because of low maintainability requirements.

2. System Integration Requirement

2.1. Popeyes/ Managers app integration, we decided a thin client-

server would be best suited because of having an easier time integrating with existing Microsoft Office software and basic camera function will reduced the cost savings from initial installation.

**Performance requirements:**

1. Speed Requirements
   1. Speed performance with thin client server architecture is more appropriate because software has high performance and it will be cheaper and easier to install.
2. Capacity Requirements
   1. Capacity requirements under thin client-server architecture would be more suitable due to higher scalability options and the system supporting an unlimited number of users. However, a thick client-server would also work because of not having an extremely large user base
3. Availability/Reliability Requirements
   1. Under a thin client-server, users would have higher reliability from multiple servers and as well as from a thick client-server or server-based architecture. However, this requirement mostly depends on the hardware and operating systems used. So, it can be either server-based, thick or thin client-server can be used.

**Security Requirements:**

1. User Authorization Requirements

1.1 User authorization should be done on server-based architecture because better of security through elimination of authorization through software, which is more prone to widespread attacks.Thin client server doesn't have effective and reduced security.

1. Access Control Requirements:

2.1 Having access control on server-based architecture would decrease chances of unauthorized users accessing the system due to an extra layer of security.

1. Encryption/Authentication requirements:

3.1 Encryption/ authentication will be done on the thick and thin-server due to advanced internet security standards used in today's cloud/internet platforms.

**Cultural/Political Requirements:**

1. Legal Requirements

1.1. Under thin client, changing legal requirements would be easier to change in the presentation logic and to disable or enable features.

Popeyes Policy Requirements

2.1. Under thin client, the changing of UMBC’s policies may be updated easier in the presentation logic as opposed to having to update every client device.

Popeyes Ethics Policy

3.1. Under thin client, the changing of popeyes ethical policies may be updated

easier in the presentation logic as opposed to having to update every client device.Either thick, server based can work in this situation.

**System Architecture Decision**

After we have appointed check mark toward the system architecture designed. We have found that our strongest choice would be a thin-client server. A thin-client server is the most cost-saving option for the other system architectures. It is also very simplified management system. It is easy to manage the system and utilize it with many different software options which can help to simplify the use of the system. It is centralized and simplifies the use of the system. It is centralized and simplifies backup of client access devices. In addition,a thin-client server has an enhanced security that is critical for our system. The system will store user’s information, and it is a top priority to keep this protected and secured. One main problem that comes with a thin-client server is its centralized nature, which can result in loss of data if the server fails. The physical security is also at risk. However, a thin-client serve will help owner to increase productivity as it has a quick start up and enables flexibility without needing a specialist to start up the system. After we compared the check mark with reasoning we have decided that thin-client is great option for your client versus thick-client and server based.

**Overall Team Matrix (AWS Custom Build)**

**\*\*Due to some time constraints and in-experience with AWS, the demo that will be shown in-class will be hosted through GL servers hosted at UMBC to meet the requirements. The end plan to host the completed application will come through AWS but until then any demos will be strictly hosted on a GL server.**

As a team we decided to go with the AWS custom build design, as it provides the most bang for our buck. The AWS service allows for us to custom build the application to the stakeholder’s requirements which allows us to customize features that may not be available with other options, such as the GoDaddy route. Also, AWS is entirely scalable from features to cost, meaning that should the option we chose with EC2 a1.large be too much hardware for our needs, the system owner can scale down and end up saving money. The EC2 service also allows for SQL support, meaning that we can create a DB for easy storage and look-up.

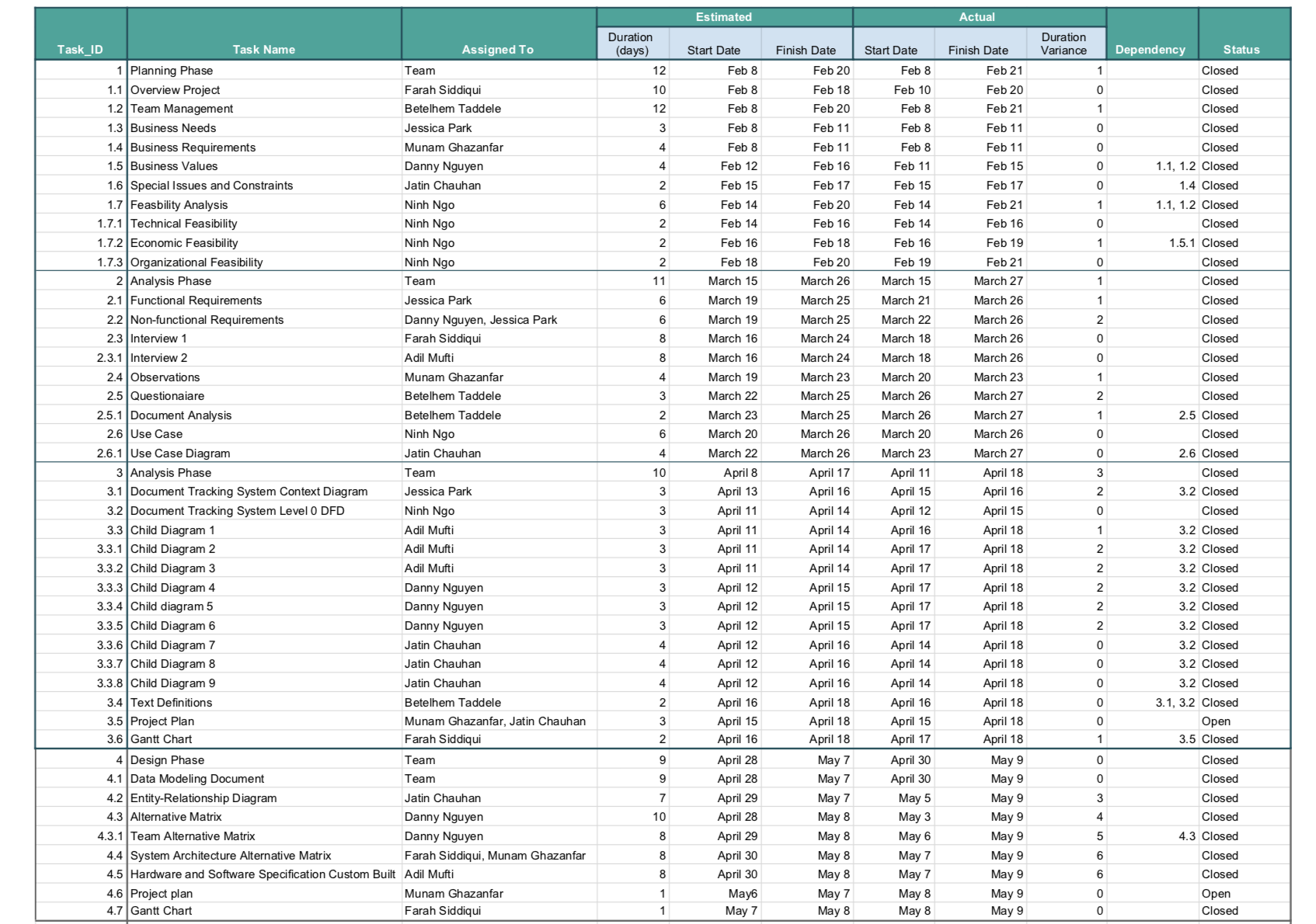
AWS is also a widely used service across many business enterprises which means that should the system owner ever find themselves with an issue they cannot solve, there are a monumental amount of help services that they can look to which also includes Amazon’s own help desk service. In terms of storage redundancy, they also provide that feature because that was a requirement from the stakeholder to keep information for extended periods of time. The cost aspect, about $263 not intaking for some added features we will need to look into, is a very small sum for the service that is provided. This is a great meeting point in terms of cost and benefits, as it is not the cheapest option but provides the one of the highest avenues for customization.

The AWS solution seems to be the perfect way to integrate the web application to the stakeholders needs at the moment and still allow for scalable options should he require more features in the future. The only issue, again, is that the stakeholder is unfamiliar with the AWS technology so at hand-off there will be a steep or medium learning curve for them. The other alternative of custom building the application and strictly using GoDaddy as a domain service is also enticing but the issue with GoDaddy comes when integrating SQL and storage features. There is an even steeper learning curve when trying to integrate all of those features that makes the lower cost option less attractive. The highly customizable features of an in-house server is great but comes at a large cost. AWS is the best option when considering all of these factors since it sits right in the middle of what this project requires.

**Figure 4.4 Hardware and Software Specification**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Client | Web Based  Server | Database Server |
| Software | Windows 10 | Linux | Oracle Database |
| Hardware | * 500 GB memory device * Intel Core i7-920 * 22-inch LED Monitor | * 500 GB disk drive * Intel Xeon Platinum 8000 | * 1-TB disk drive * RAID |
| Network | * 1 Gbps bandwidth * Popeyes Wifi | * GL Server | * GL Server |

**Project Plan**

****

**Gantt Chart**

