

# Technology Research Document

Name: Tony Jiang

Semester: 6

Class: RB04

## Contents

Introduction .....	3
Technology for backend (individual).....	3
Database selection (individual).....	3
CAP theorem .....	3
Polyglot persistence .....	4
Database selection for service .....	5
User service .....	5
Song service .....	5
Text file store (group).....	5
Cloud technology (individual).....	7
Reason for deploying to cloud.....	7
Cloud function vs Kubernetes cloud .....	8
Cloud provider .....	8
Pricing.....	8
How to deploy manually for Golang .....	9

# Introduction

This document contains all the research I've done on the technology and methods I use, along with the reasons for my choices. This way, I can showcase to the teachers that I have explored the options and justified my decisions. This research can be for the individual project or the group project.

## Technology for backend (individual)

This semester, I have the opportunity to explore a new programming language. A friend recommended that I try Golang, so I did some research. Initially, I was planning to use Java because of its vast library options, built-in security features, high scalability, and strong support for microservices.

However, Golang (Go), which was designed by Google, also offers several appealing features. It has a simple and easy-to-read design, fast compilation, strong support for microservices, and is excellent for cloud-native development. Additionally, it can easily scale horizontally and is highly favored for DevOps and infrastructure tasks. The only problem is that I need to learn how to code in Golang.

The obvious choice is Golang because it adheres to all the learning outcome of this semester.

## Database selection (individual)

Choosing the right database for a project can be challenging. To make this decision, I use the CAP theorem and its reference Canvas.

### CAP theorem

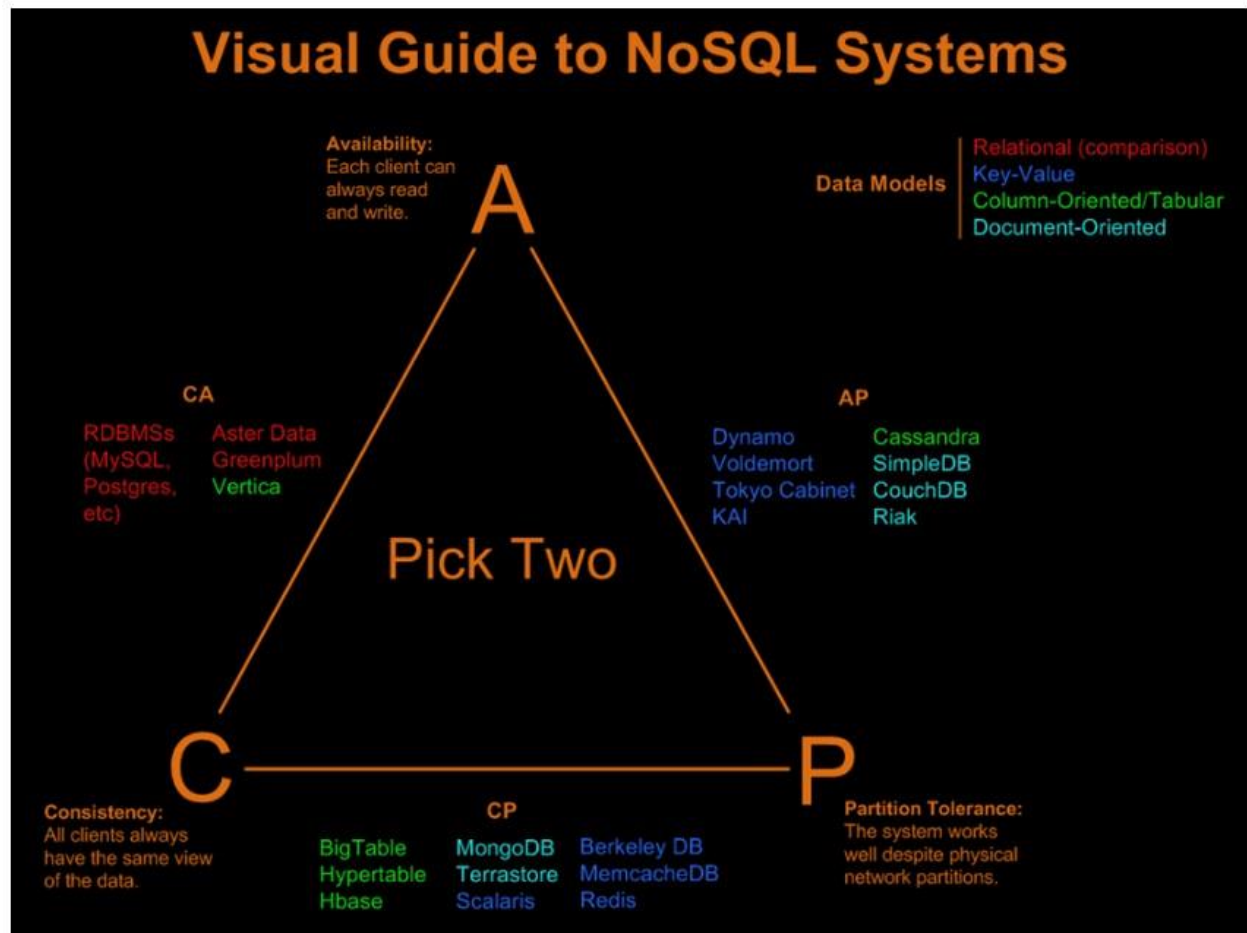
The CAP theorem is a fundamental concept in distributed database systems. It states that it is impossible for a distributed data store to simultaneously guarantee all three of the following properties:

- Consistency (C): Every read receives the most recent, correct, and up-to-date information, regardless of which server (or node) you ask.
- Availability (A): Every request (read or write) receives a response, even if it doesn't guarantee that the response contains the most recent write.
- Partition Tolerance (P): The system remains operational despite network partitions or communication breakdowns between nodes.

According to the CAP theorem, you can only guarantee two out of these three:

- CP (Consistency + Partition Tolerance): The system remains consistent and tolerates network partitions, but this comes at the cost of availability—some requests may fail.

- AP (Availability + Partition Tolerance): The system prioritizes responsiveness and handles network issues, but consistency may be sacrificed, meaning you might not always receive the latest data.
- CA (Consistency + Availability): The system ensures both accurate, up-to-date data and immediate responses, but it cannot tolerate network failures or partitions.



Based on the CAP theorem, my project should prioritize AP (Availability + Partition Tolerance) to ensure quick response times and resilience to network issues. My users prefer faster responses over strict data consistency since they are playing a game, and they value immediate feedback on their answers more than having perfectly consistent data. Therefore, my system will need an AP-focused database. A reference to AP databases is provided in the image above.

## Polyglot persistence

In microservices architecture, it's common to use Polyglot Persistence, where each microservice can use the database best suited to its specific needs. You don't need to rely on a single type of database for the entire system. Different microservices can use different databases based on their requirements.

## Database selection for service

Here is the database selection for the service that are currently implemented for the application.

### User service

For the user service, I want to prioritize data accuracy and consistency. A relational database like PostgreSQL or MySQL is well-suited for this purpose, as these databases ensure that user data—such as account details, login credentials, and profile information—remains consistent and accurate.

PostgreSQL: Known for its robustness and support for complex queries, it is ideal if you anticipate needing to manage intricate relationships, such as user roles and permissions.

MySQL: A faster, reliable option for handling straightforward relational data and transactional workloads.

For my user service, I have chosen PostgreSQL because of its robustness and ability to handle complex relationships. Instead of hosting PostgreSQL locally, I will use a cloud-based solution to simplify future deployment. Using a cloud-hosted database eliminates the need to configure connections between my application and a remote PostgreSQL instance later on, making the setup more efficient from the start.

The PostgreSQL cloud provider I am using is Supabase, which offers a free tier. This includes 500 MB of storage, 2 GB of bandwidth, and 50 concurrent connections. While exceeding these limits incurs additional costs, the free tier is sufficient for my current needs.

### Song service

For the song service, MongoDB is an ideal database due to its flexible data structure and its ability to handle frequent read and write operations efficiently. MongoDB's document-based storage is well-suited for storing song metadata, such as the title, artist, genre, and other attributes, while also supporting frequent CRUD operations. Its schema flexibility allows for seamless modifications as the song data evolves over time.

MongoDB supports fast and flexible CRUD operations and scales horizontally, which is advantageous as the volume of song data grows. Additionally, it integrates easily with message brokers for sending events (e.g., CRUD operation events).

I have chosen MongoDB Atlas for cloud storage instead of hosting it locally. Using a cloud-based solution eliminates the need for future configuration and setup, saving time and effort. It's better to address this early in the project to streamline future development and deployment.

## Text file store (group)

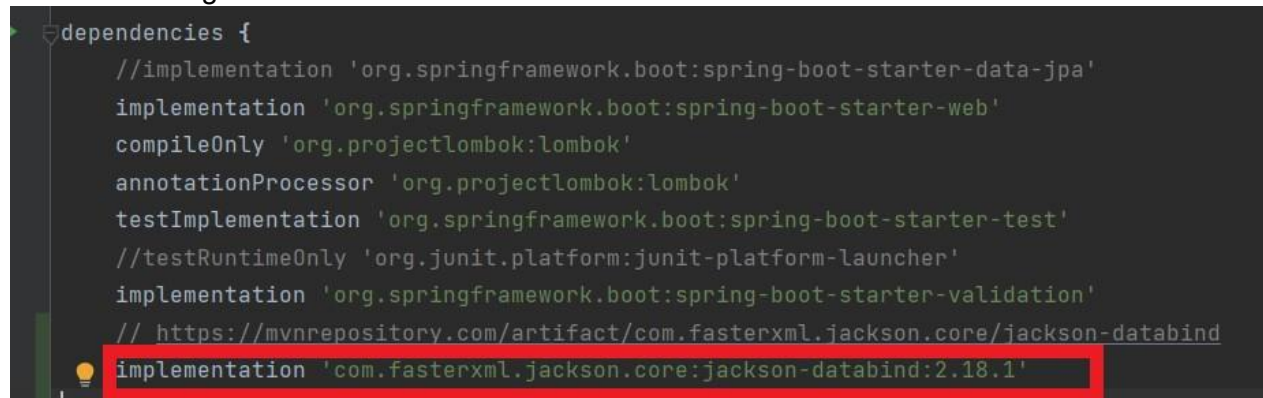
I was tasked with creating a text file storage solution for the botlist service for the group. Since I had no prior knowledge or experience in this area, I conducted some research on how to implement it. The language used to create the botlist service is Java, and the text file can be stored in JSON format.

I explored possible solutions for storing data in a text file using Java and identified two dependencies that could meet my requirements:

the Gson library and the Jackson library.

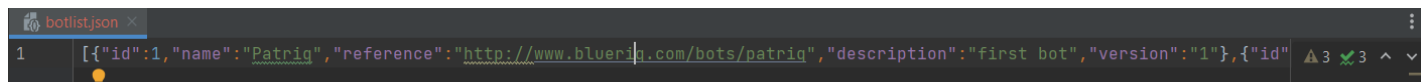
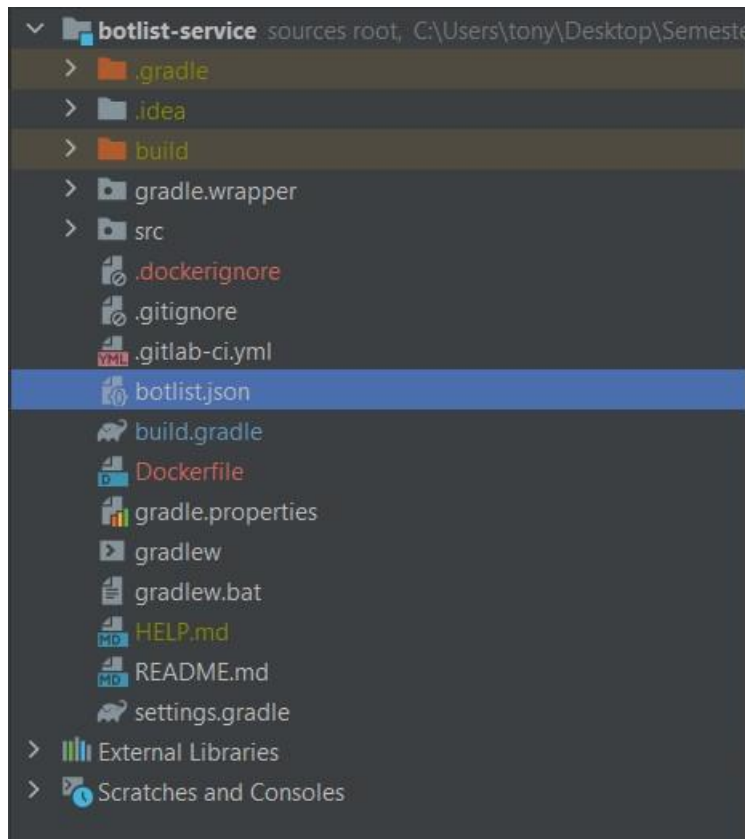
I chose the Jackson library due to its advanced features, better performance, and support for polymorphism. This choice also allows for future scalability, ensuring high performance and accommodating larger datasets if the company decides to expand.

While the Gson library is simpler and more beginner-friendly, I opted for Jackson to better suit the project's needs. All you need to do is add the Jackson dependency in build.gradle, and you can start coding.



```
dependencies {  
    //implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    compileOnly 'org.projectlombok:lombok'  
    annotationProcessor 'org.projectlombok:lombok'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
    //testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
    implementation 'org.springframework.boot:spring-boot-starter-validation'  
    // https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind  
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.18.1'
```

The text file is stored in the root folder of the botlist service. If the file does not exist, the system will automatically create it when a botlist is added. However, the file does not track IDs, as it lacks an auto-increment mechanism like a database, so an auto-increment feature will need to be implemented manually.



## Cloud technology (individual)

For the cloud technology, I'm planning to use cloud functions. While I have more experience with Kubernetes, I want to explore cloud functions as they are new to me. From my understanding, with cloud functions, instead of deploying an entire microservice to the cloud, you can deploy individual functions in a serverless manner. However, I'm not yet familiar with how they are deployed.

## Reason for deploying to cloud

I am using the cloud for my project because it allows me to scale my application depending on the number of users. My application is a music guessing game, where users challenge themselves to guess the daily song or guess random songs. Since the application should handle a large number of users, deploying it to the cloud is ideal for scaling the project.

## Cloud function vs Kubernetes cloud

The main difference between Kubernetes and cloud functions lies in their deployment and management models.

**Kubernetes:** With Kubernetes, you deploy and manage entire services, such as applications or containers, on the cloud. This typically involves setting up and maintaining infrastructure for scaling, availability, and load balancing. While Kubernetes provides robust control over the service lifecycle, it can lead to higher operational overhead and costs, as you're responsible for managing the infrastructure. The cost of hosting a service on Kubernetes begins as soon as the service is deployed and continues to accumulate based on how long your containers or pods are running.

**Cloud Functions:** In contrast, cloud functions are serverless. This means you deploy individual functions (specific pieces of code designed to perform tasks) rather than entire services. These functions are event-driven and run only when triggered—by user interactions, events, or other stimuli. Costs are determined by the number of invocations (triggers) and the function's execution time, rather than continuous uptime like in Kubernetes. This pay-as-you-go model often makes cloud functions more cost-effective for smaller, event-driven tasks.

In summary: Kubernetes is designed for deploying and managing entire microservices on the cloud. For example, my song service, which handles CRUD operations for songs and manages song data, would be well-suited for Kubernetes. It allows me to manage multiple services, scale efficiently, and have fine-grained control over how my application is deployed and managed.

Cloud Functions are designed for simpler, event-driven tasks, such as my Song Suggestion service. This service only needs to process user input and suggest songs based on that input, making it ideal for a function-based architecture. With cloud functions, you pay for what you use, making it a more cost-effective option for smaller services.

While cloud functions can serve as an option for simple CRUD operations, deploying a full microservice (like a song service) with various interactions and data storage is often better suited to containers or Kubernetes for better scalability and management.

## Cloud provider

For the cloud provider I can use Google, AWS or Azure. Azure is out of the question because I'm using Golang, I have to configure a lot of things for Golang. Google and AWS is ideal because of their ease of use, extensive support, and ecosystem integrations.

## Pricing

Google offers a free tier with \$300 in credits valid for 90 days. After that, you are billed for usage. The free tier includes 180,000 vCPU-seconds and 360,000 GiB-seconds per month. Google bills execution time in vCPU-seconds and GiB-seconds. Beyond the free tier, the rates are \$0.000018 per vCPU-second and \$0.000002 per GiB-second. These rates can get more cost-effective with committed-use discounts for long-term, high-volume usage.

AWS provides a free tier with 7.5 billion GB-seconds per month, which is generous and suitable for low-cost startups. After exceeding the free tier, AWS bills based on GB-seconds (a



combination of memory allocated and execution time). AWS charges \$0.0000133334 per GB-second initially, with rates decreasing at higher usage volumes.

For testing purposes I'm using AWS, AWS is ideal due to its generous free tier. However, for long-term, high-volume scenarios, Google can be more cost-effective, especially with committed-use discounts.

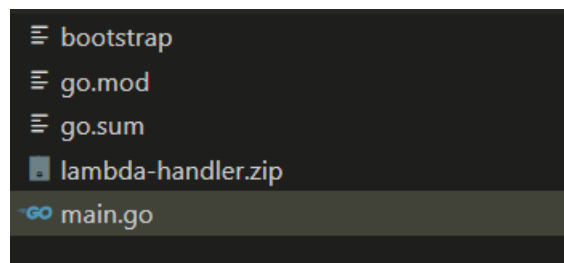
## How to deploy manually for Golang

To test this, I use my song service to deploy a Golang function to AWS Lambda, I started by importing the AWS Lambda Go SDK into my project using the package "github.com/aws/aws-lambda-go". I then created a handler for Lambda, utilizing my CreateSong function as an example. To connect the handler to the Lambda runtime, I added it to the lambda.Start() function, ensuring that the code was ready for deployment.

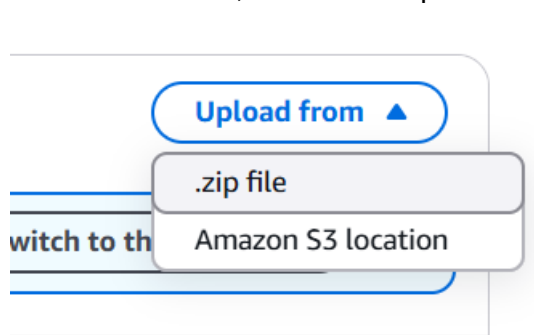
```
func lambdaHandler(ctx context.Context, req events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
    var newSong Song
    if err := json.Unmarshal([]byte(req.Body), &newSong); err != nil || newSong.Title == "" || newSong.Artist == "" || newSong.Genre == "" {
        return events.APIGatewayProxyResponse{StatusCode: http.StatusBadRequest, Body: `{"message": "Invalid input"}`, nil}
    }
    response, _ := json.Marshal(createSong(newSong))
    return events.APIGatewayProxyResponse{StatusCode: http.StatusCreated, Body: string(response)}, nil
}

func main() {
    lambda.Start(lambdaHandler)
}
```

Next, I built a binary file from my main.go file. Since AWS Lambda requires the binary to be named bootstrap, I made sure to name it accordingly. I used the following command to create the binary: `$env:GOOS="linux"; $env:GOARCH="arm64"; $env:CGO_ENABLED="0"; go build -o bootstrap main.go`. This step ensured that the binary was compatible with Lambda's runtime environment. Additionally, I configured my environment file to match the environment instance I created in the Lambda UI. After creating the binary, I zipped it into a .zip archive and uploaded it to AWS Lambda.



In the Lambda UI, I used the “Upload from” option and selected the .zip file for upload.



Once the file was uploaded, I tested the function within the Lambda UI. I noticed that the request body format for Lambda tests differed slightly from Postman, so I adjusted the test input accordingly. The function ran successfully, and I received a response confirming its execution.

### Event JSON

```
1 {  
2   "body": "{\\"title\\": \\"Image\\", \\"artist\\": \\"Tina\\", \\"genre\\": \\"Pop\\"}"  
3 }
```

 **Executing function: succeeded** ([logs](#))

 **Details**

The area below shows the last 4 KB of the execution log.

```
{  
  "statusCode": 201,  
  "headers": null,  
  "multiValueHeaders": null,  
  "body": "{\\"id\\":\\"1\\",\\"title\\":\\"Image\\",\\"artist\\":\\"Tina\\",\\"genre\\":\\"Pop\\"}"  
}
```

**Summary**

To make the function accessible, I added an API Gateway as a trigger. This setup generated a URL for the function, allowing me to send requests. In this case, I configured the API Gateway to handle POST requests. Finally, I tested the URL in Postman to verify that the function was triggered correctly and performed as expected.

☐ | Trigger



**API Gateway:** [CreateSongFunction-API](#)

arn:aws:execute-api:eu-central-1:730335556224:hy9yemcj2d/\*/CreateSongFunction



API endpoint: <https://hy9yemcj2d.execute-api.eu-central-1.amazonaws.com/default/CreateSongFunction>

► Details

[HTTP](#) Individual Project Sem 6 / Test Cloud / Create song

**POST** ☐ <https://hy9yemcj2d.execute-api.eu-central-1.amazonaws.com/default/CreateSongFunction>

Params Authorization Headers (9) **Body** ☒ Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ☐

```
1 {  
2   "title": "Image",  
3   "artist": "Ben",  
4   "genre": "Pop"  
5 }
```

**Body** Cookies Headers (5) Test Results

**201 Created** • 92 ms • 236 B •

Pretty Raw Preview Visualize Text

```
1 {"id":"2","title":"Image","artist":"Ben","genre":"Pop"}
```

That's the process I followed to deploy a Golang function to AWS Lambda. By using the Lambda UI and some manual steps, I successfully deployed, tested, and triggered the function.