RESEARCH DOCUMENT

[Document subtitle]

OCTOBER 8, 2024

TONY JIANG
RB04

Contents

Introduction

This research document addresses my main question by breaking it down into sub-questions. By answering these sub-questions, I aim to determine a conclusive answer to the main question, which pertains to my individual project: building a web-based music guessing game using Golang.

Research questions

Main question:

How can I design a scalable web-based music guessing game using Golang within a microservice architecture?

Sub-questions:

1. **How should microservices be structured in Golang for optimal scalability and maintainability as the user demand increase?**
    - **Strategy: Library**
    - **Methods: Best good and bad practice – Design pattern research**

2. **How should the security of the music guessing game be validated?**
    - **Strategy: Library – Lab**
    - **Methods: Literature study - Security test**

3. **How can CI/CD pipelines be set up for deploying a microservice-based game using Golang?**
    - **Strategy: Library – Workshop**
    - **Methods: Literature study - Community research – Prototyping**

4. **What are the best practices for deploying Golang microservices using serverless cloud functions?**
    - **Strategy: Library – Workshop**
    - **Methods: Best good and bad practice – Prototyping**

5. **What tools can be used to monitor the scalability of microservices?**
    - **Strategy: Library – Lab**
    - **Methods: Literature study – Community research – Non-functional test**

6. **How should data be managed across distributed microservices to ensure consistency, reliability, and scalability?**
    - **Strategy: Library**
    - **Methods: Literature study - Best good and bad practice**

Sub-questions

1. How should microservices be structured in Golang for optimal scalability and maintainability as the user demand increase?

To start, it's important to understand what microservices are, whether Golang can be structured in a microservice architecture, and how we can scale and maintain these microservices effectively.

What are microservices?

Microservices are small, independent services within an application that each handle specific functions, collectively forming the entire application. Each service can be built, deployed, and scaled independently, making the architecture flexible and easier to manage. This approach is commonly used in growing applications and enterprise environments.

Golang can be designed in a microservice architecture. There are tools available that support Golang in creating and managing microservices, making the architecture easier to maintain and scale.

https://aws.amazon.com/microservices/

https://www.redhat.com/en/topics/microservices/what-are-microservices

https://www.bacancytechnology.com/blog/golang-microservices-architecture

How to divide your application into microservices?

To divide your application into microservices, start by identifying its functionalities and grouping them based on similarities. Assign a service name to each group, keeping in mind that a microservice can consist of a single function.

For scalability, microservices should ideally be decoupled, meaning they can function independently and communicate without relying on the internal workings of other services. However, certain requirements may lead to tightly coupled services, where one service depends on another to operate, which can introduce configuration challenges.

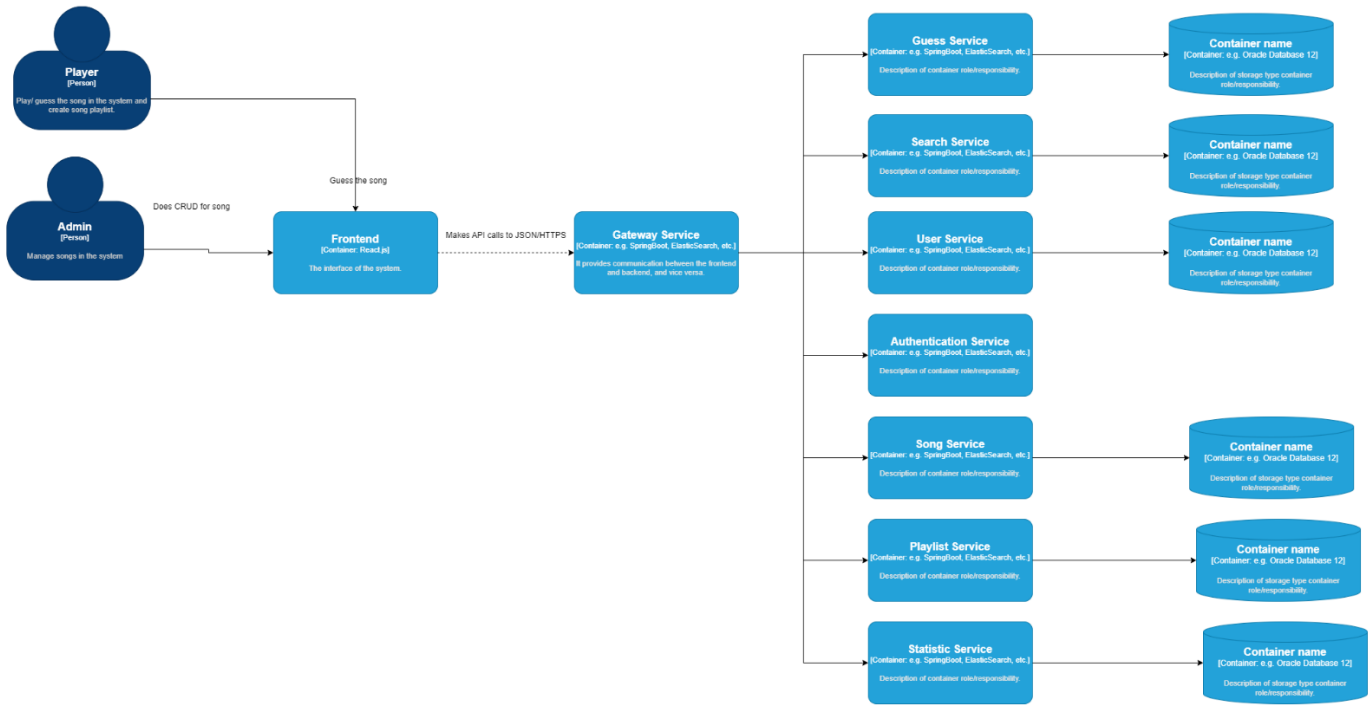
Each microservice should have its own database to avoid inter-service dependencies for data retrieval. This approach helps reduce response times and ensures a more resilient system.

https://martinfowler.com/articles/break-monolith-into-microservices.html

https://learn.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis

Result

As a result of what I understand, here is the first draft of the microservices architecture design I created for a web-based music guessing game.



It looks promising, but some services still lack necessary connections with others. For example, the Playlist Service needs data from the Song Service to include relevant songs. However, I don't want the Playlist Service to depend directly on the Song Service, as that would undermine the purpose of decoupling services and maintaining their own databases.

What is scalability in a microservice architecture?

Scalability in a microservice architecture refers to designing and configuring your services and infrastructure to efficiently handle increased loads. This can be achieved using tools and strategies like:

- **Kubernetes:** Deploy your services using Kubernetes to enable horizontal scaling by creating more instances of a service. It can be deployed on cloud-based platforms, whether server-based or serverless, for example cloud functions. However, it does costs money to maintain your services.

- **Docker:** Containerize each service to streamline deployment and ensure consistency across environments.
- **Database Management:** Each service should have its own database, with the type of database chosen based on the service's specific requirements.
- **Monitoring and Observability:** Implement monitoring tools to track the performance of your services, identify potential issues, and make scaling decisions effectively.
- **Asynchronous Communication:** Use message brokers like Kafka or RabbitMQ for asynchronous communication to decouple services and improve scalability.

https://www.couchbase.com/blog/scaling-microservices/

https://www.liquidweb.com/blog/microservices-scalability/

How would the microservice communicate with each other?

There is various way of how to microservice communicating with each other. It depends on the systems requirement like performance, consistency or scalability.

The communication options are:

- **Synchronous Communication (Request-Response)**
  - **HTTP/REST:** This approach is where one service makes an HTTP request to another service, often using JSON as data format. This method is simple but introduces latency and may create tightly coupled services.
  - **gRPC:** It's a high performance from HTTP/Rest with low latency requirements.
  - **GraphQL:** Allows clients to query exactly the data they need from multiple services in a single request. It is more flexible than HTTP/REST but can introduce complexity in certain use cases.
  - **WebSocket:** Used for real-time, bidirectional communication between services. Suitable for scenarios like streaming or when both services need to push updates.
- **Asynchronous Communication (Event-Driven or Message-Based)**
  - **Message Queues:** Services communicate by sending messages to queues or topics. This decouples services and improves reliability. Examples include:
    - Kafka
    - RabbitMQ
    - Amazon SQS
  - **Event-Streaming:** Services publish events (like order completed, payment received) to a message broker, and interested services can consume those events asynchronously. This is useful in event-driven architectures.
  - **Pub/Sub:** Services publish messages to topics, and subscribers (other services) receive them. Systems like Google Pub/Sub, Redis Streams, or NATS are commonly used.
- **Hybrid Communication**

- **Hybrid HTTP + Event-Driven:** Some systems mix both synchronous and asynchronous communication patterns. For example, services might use HTTP for real-time communication (e.g., user authentication) but use event-driven messaging (e.g., Kafka or RabbitMQ) for less time-sensitive operations (e.g., notifications).
- **Service Mesh:** Tools like Istio or Linkerd manage service-to-service communication by abstracting traffic routing, load balancing, and security (e.g., mTLS) away from the application logic.

https://www.geeksforgeeks.org/microservices-communication-patterns/

https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71

https://identio.fi/en/blog/building-a-robust-microservice-architecture-understanding-communication-patterns/

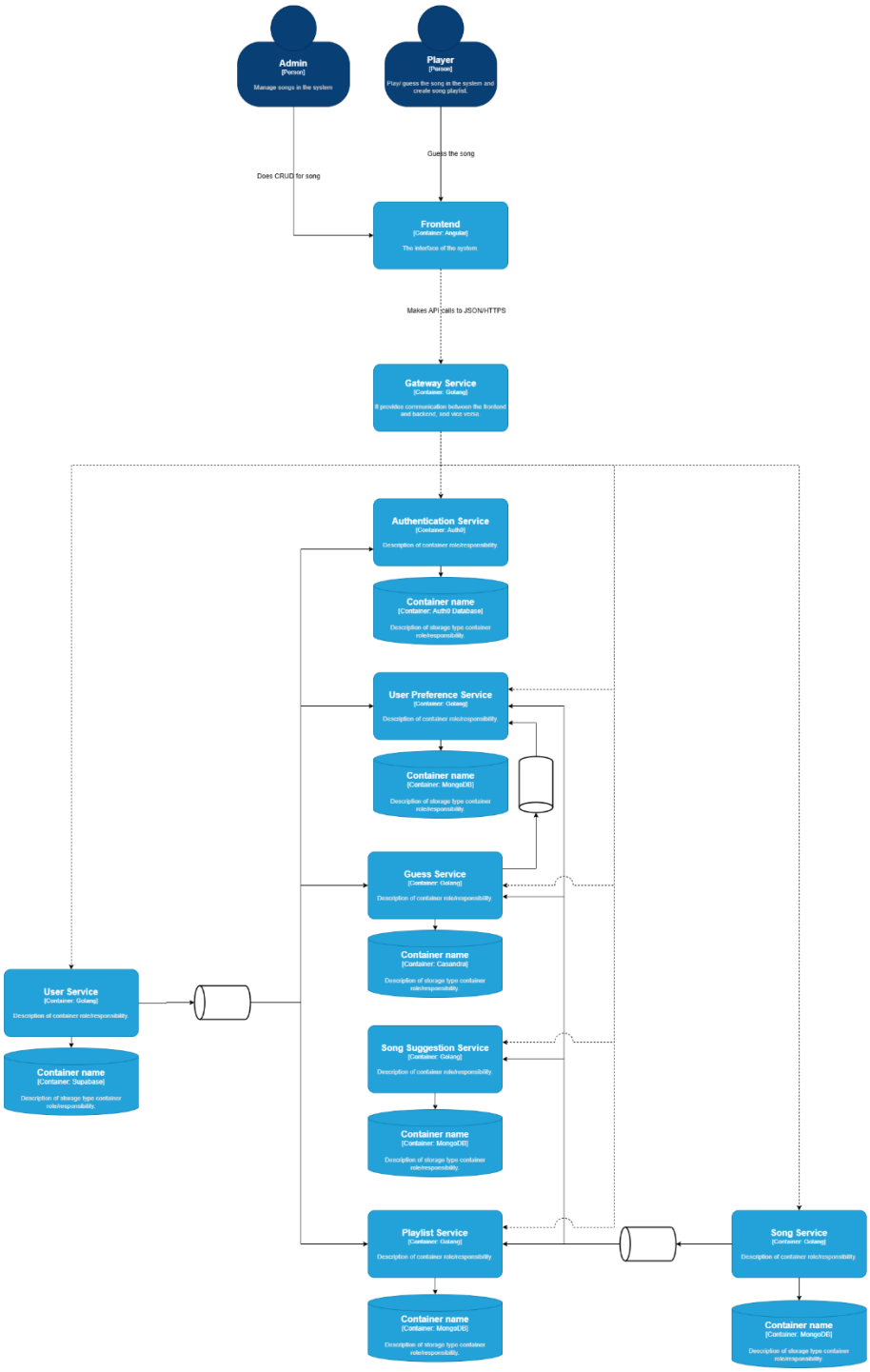
Based on the information I've gathered, I plan to use a hybrid communication model in my microservice architecture, combining HTTP/REST requests and message queues. The success of this approach will depend on designing it to ensure scalability, high performance, and some level of consistency.

Since this is a web-based game, consistency is less critical; players prioritize performance, and the system must handle high traffic during peak hours. HTTP/REST will be used primarily for the gateway service, while all other services will communicate with each other through a message queue.

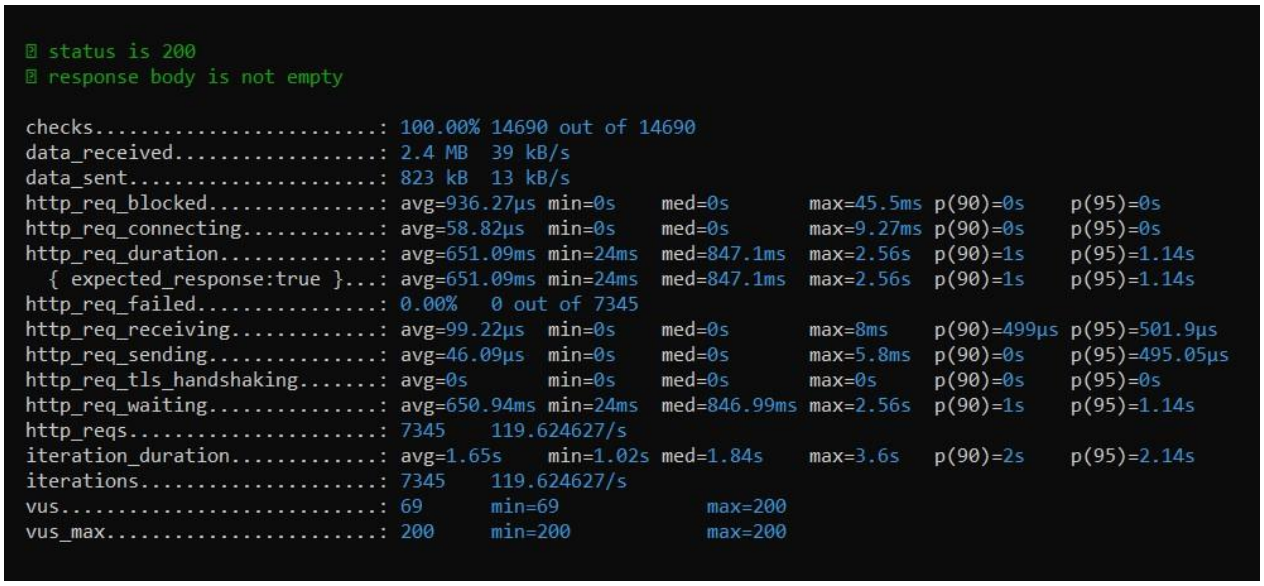
I also need to consider GDPR compliance when handling player data. For example, if the Playlist Service uses player data, such as the player ID and username, to associate playlists with users, it must adhere to GDPR guidelines. This means that if a player deletes or updates their information in the User Service, the data must also be deleted or updated in other services, like the Playlist Service, that rely on it.

Result

Here is the result of the final version of the microservice architecture for my project.



Now it's a bit better. For more details on what each service does, you can find the information in my Technical Design Document, specifically in the "Microservice Architecture" section. I performed load testing, and the results look promising. I didn't have all the services set up due to time constraints, but I tested the song-suggestion service using K3d to simulate the production environment and demonstrate how it works.



For the load test, I used K6. Achieving around 7,300 requests is not bad, but due to the limitations of my laptop's resources, I couldn't test further. Ideally, the service should be deployed to the cloud for a more accurate load test. However, I now have a better understanding of how to perform load testing.

What is maintainability in microservice architecture?

Maintainability in microservice architecture refers to how easy it is to update, fix, or add a service without hindering, breaking, or causing downtime for other services. To ensure maintainability, it is crucial to keep the architecture agile and scalable over time.

To achieve maintainability, the following practices are essential:

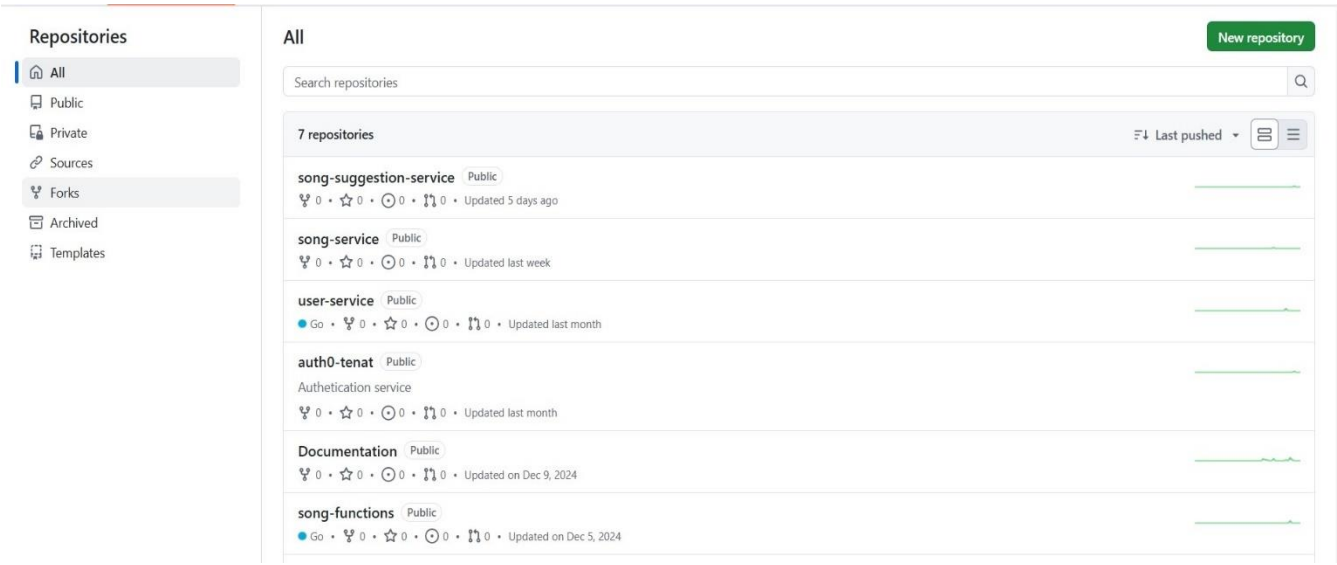
- **Loosely Coupled Services:** Services should not depend directly on each other but should communicate through well-defined interfaces or protocols.
- **Clear Boundaries and Responsibilities:** Each microservice should have a well-defined purpose, reducing overlap and avoiding confusion about where functionality is implemented.
- **Automated Testing:** Services should have robust unit, integration, and end-to-end tests to ensure that changes do not break functionality or interactions with other services.
- **Version Control:** Each service should have its own repository and versioning system, allowing changes to be rolled back without impacting other services. Unlike the monolithic approach of placing all services in a single repository, each service is maintained in its own separate repository.
- **Consistent Deployment Pipelines:** CI/CD pipelines should ensure services can be deployed or rolled back independently, with automated tests verifying changes before deployment.
- **Decentralized Data Management:** Each service should manage its own data to minimize cross-service dependencies and simplify schema changes or updates.

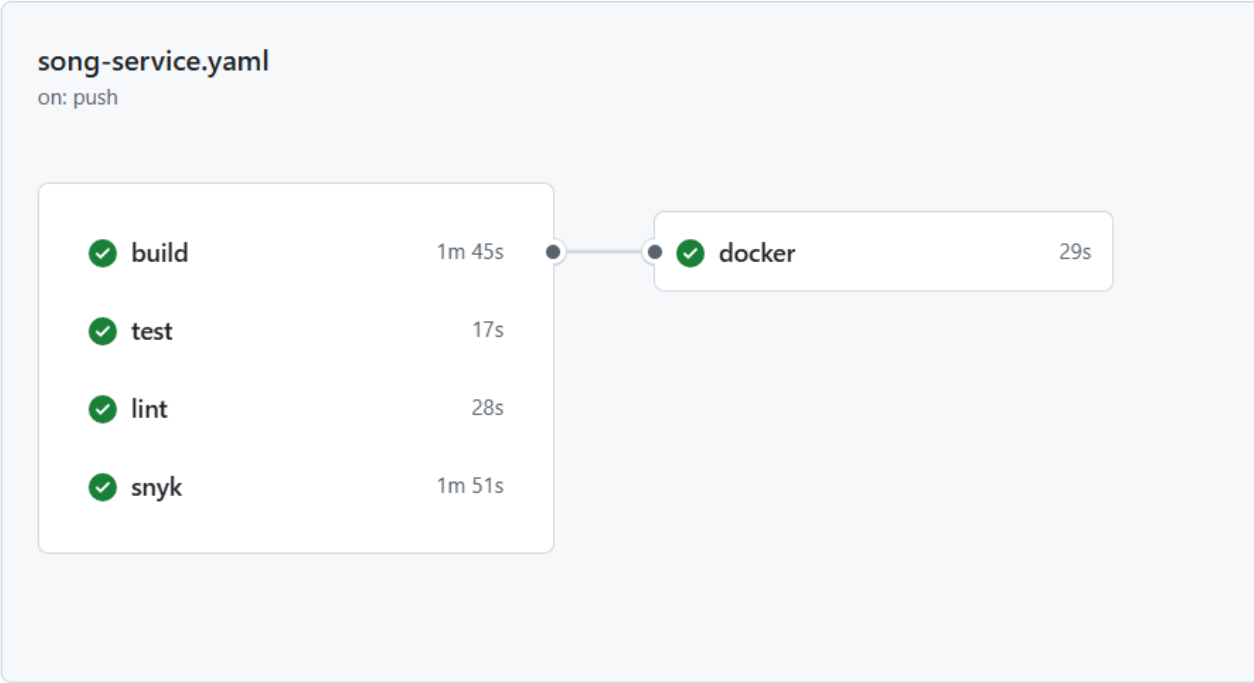
https://www.index.dev/blog/implementing-microservices-for-scalability-maintainability

https://gnithyanantham.medium.com/microservice-architecture-design-patterns-the-secret-to-building-scalable-resilient-and-82747886c017

https://www.linkedin.com/pulse/understanding-importance-microservices-maintenance-seminda-rajapaksha-yqcec

In my project, all the services have their own repositories, unlike a monolithic setup where all microservices are kept in a single repository. Each repository includes CI/CD pipelines to continuously build, test, and deploy the services. Additionally, each microservice has its own database to avoid dependencies on other services, ensuring they are loosely coupled. This can be seen in the final version of the microservice architecture.





The deployment part is not yet included at the time of writing this, but it will be added in the final project.

Conclusion

In Golang, you can structure a microservice architecture just like in any other programming language, as Golang provides robust support for all necessary features. To scale optimally in a microservice architecture, you should:

- Deploy your project to Kubernetes to enable horizontal scaling by creating instances of your services.
- Decouple services, ensuring each has its own database to avoid dependencies.
- Containerize services using tools like Docker for consistent deployment.
- Use asynchronous messaging systems like Kafka or RabbitMQ to facilitate communication between decoupled services.

For maintainability, it is essential to:

- Divide and decouple services into separate repositories to manage them independently.
- Implement CI/CD pipelines for each service to continuously build, test, and deploy them automatically.
- Each service should have their own database.

2. How should the security of the music guessing game be validated?

Security refers to the practice of protecting systems, data, and resources from unauthorized access, damage, or theft. It covers a wide range of measures, strategies, and technologies aimed at ensuring the confidentiality, integrity, and availability of information and systems.

What is security for a microservice architecture?

For a microservice architecture it refers to the practices, technologies, and strategies used to protect the various components of a microservices-based system, ensuring that each service, communication channel, and the overall system remains secure, reliable, and resistant to attacks. Since microservices are distributed, decentralized, and often interact with other services over the network, security in such architectures becomes more complex compared to monolithic systems.

The music guessing game is built on a microservice architecture. The key areas of security to focus on are:

- **Authentication and Authorization:** Use JWT to validate or grant access to users based on their roles. You can also use a third-party authentication provider like Auth0 or OpenID to enable proper authentication and authorization.
- **Input Validation and Data Sanitization:** Implement strict validation checks for user input and sanitize it to prevent SQL injection, cross-site scripting (XSS), and other injection attacks.
- **Encryption and Protection of Secrets:** Encrypt sensitive data, such as user passwords, and protect any secrets (e.g., API keys, client secrets, or credentials for basic authentication). It's important not to hard-code API keys. Use tools like HashiCorp Vault, Microsoft Azure Key Vault, or Amazon KMS to store and manage secrets securely.
- **API Security:** Use an API Gateway to centralize security concerns like authentication and authorization. You can also implement Cross-Origin Resource Sharing (CORS) if your application communicates with external domains or third-party services, ensuring only trusted resources can access your application.
- **Logging and Monitoring:** Log user login activity and monitor any abnormalities in API requests.
- **Secure Coding Practices:** Perform code reviews and use automated tools to ensure the code follows security best practices.
- **Vulnerability Scanning:** Implement automated tools to scan for vulnerabilities, such as outdated dependencies and security issues.

https://www.okta.com/resources/whitepaper/8-ways-to-secure-your-microservices-architecture/

https://www.styra.com/blog/microservices-security-fundamentals-and-best-practices/

https://www.designgurus.io/answers/detail/how-do-you-ensure-security-in-a-microservices-architecture

OWASP top 10

The security can be also validated using the OWASP Top 10 (2021), a widely recognized standard for identifying and addressing the most critical web application security risks. I chose the 2021 version because it includes the most recent updates and highlights key security considerations. This enables me to validate the application against these standards and generate a report to ensure compliance with OWASP Top 10 recommendations. The OWASP Top 10 is particularly important for microservice architectures due to the distributed and often complex nature of microservices.

These are the OWASP Top 10 security risks, listed in order of priority, from the most important to the least likely to occur. The first is the most critical to address, while the last represents those with lower incidence rates.

Every sprint, I review the report to validate the security risks I addressed during the sprint and add new security measures for the risks I plan to cover in the next sprint. The report should include the following: the objective of what will be done, an explanation of the risk, the acceptance criteria, the method of validation, the outcome of the validation, and notes on what is missing or what further actions can be taken. All of this is included in my OWASP Top 10 report.

Conclusion

An OWASP Top 10 report is particularly important for microservice architectures due to the distributed and often complex nature of microservices. A report should be generated every sprint to validate the security risks of the music guessing game. Additionally, new security measures should be implemented in the next sprint to address the identified risks, which can be validated at the end of the sprint.

3. How can CI/CD pipelines be set up for deploying a microservice-based game using Golang?

CI/CD stands for Continuous Integration and Continuous Deployment, and it refers to a set of practices used in modern software development to automate the process of integrating code changes, testing them, and deploying them to production environments.

Continuous Integration (CI) involves integrating code changes and automatically testing whether they have been correctly implemented. This includes build tests, code coverage, automated security scans, integration tests, and end-to-end tests.

Continuous Deployment (CD) is the next step after CI, where, once all automated tests pass, the code is automatically deployed to production. This can include pushing the code to Docker Hub and deploying it to the cloud.

https://www.redhat.com/en/topics/devops/what-is-ci-cd

https://about.gitlab.com/topics/ci-cd/

CI/CD tasks for Golang

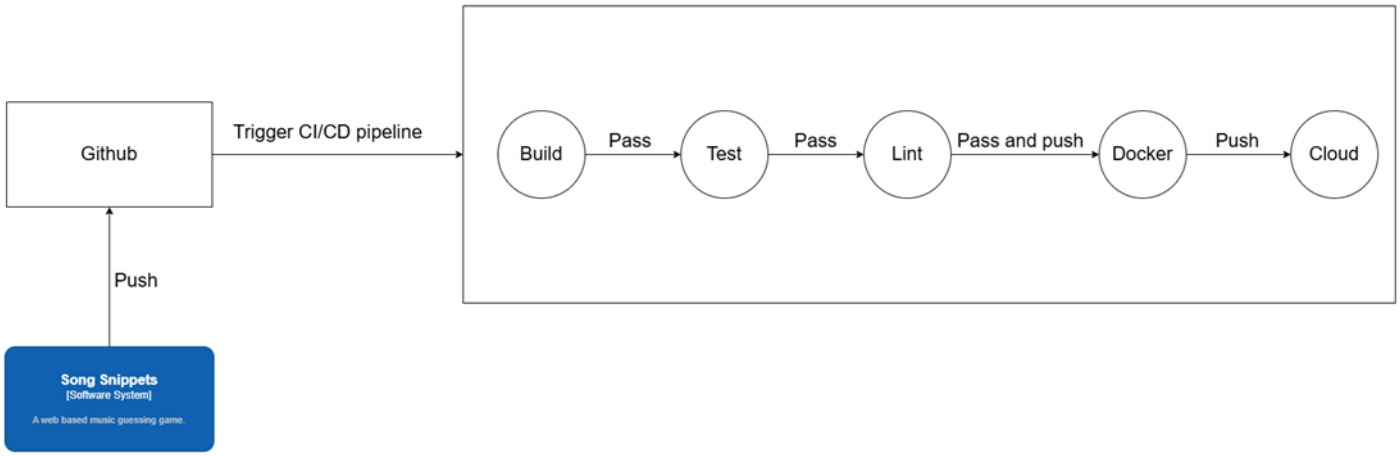
The essential tasks required for each service in the CI/CD pipeline for Golang are:

- **Build**: Builds the project to ensure there are no issues during compilation.
- **Test**: Runs unit tests to verify that code changes have been correctly implemented. If the tests pass, it means the code works as expected and no unintended changes were introduced. If the tests fail, it indicates changes to the code that need to be fixed.
- **Code Analysis**: Uses tools to ensure the code follows best practices. For Golang, you can use a linting tool for code analysis.
- **Dependency Scan**: Checks for outdated dependencies or security vulnerabilities in the project. Tools like Snyk can be used for this purpose.
- **Integration Testing**: Tests the integration between your service and external systems, such as a database or another service, via REST APIs or message queues. These tests typically involve mocking the external systems to send or receive information.
- **End-to-End Testing**: Simulates real user interactions on the frontend to ensure the entire system behaves as expected.
- **Build Docker Image**: Dockerizes the application and automatically pushes the updated image to Docker Hub whenever the project is updated.
- **Deploy to Cloud**: Automatically deploys the updated project to the cloud.
- **Monitoring**: Monitor the application health and quickly identify the issues after the deployment.

https://codefresh.io/learn/ci-cd-pipelines/ci-cd-process-flow-stages-and-critical-best-practices/#:~:text=In%20the%20field%20of%20DevOps,deploy%20the%20code%20to%20production.

What essential job/ task is good have in a CI/CD pipeline?

I used GitHub for version control and GitHub Actions to set up the CI/CD pipeline. You can find more details about how I designed my CI/CD pipeline in my DevOps Document. Below is the initial version of the design.



The pipeline is functional but still missing some tasks and I need to optimize the CI/CD pipeline to reduce the time it takes to execute all the tasks.

Optimize CI/CD pipeline

This is to reduce the time it takes to execute the tasks on the pipeline.

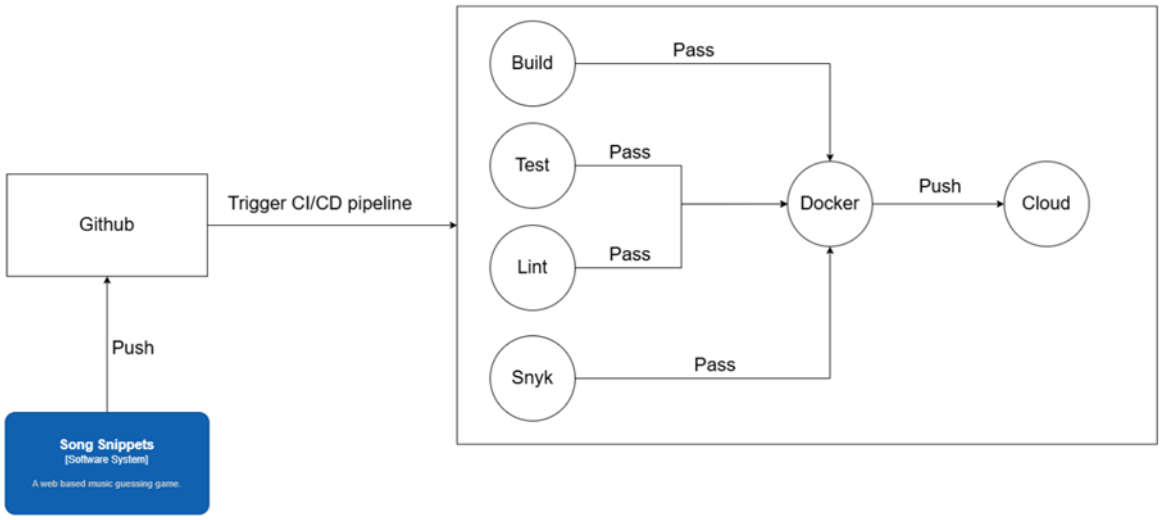
Here is how you would optimize the CI/CD pipeline:

- **Parallelize Tasks:** Spit your task to run in parallel with each other.
- **Cache Dependencies and Artifacts:** Store downloaded dependencies or build artifacts to avoid fetching or rebuilding them repeatedly.
- **Run Tests Selectively:** Only run the tests relevant to the changes made.
- **Use Faster Infrastructure:** Upgrade your build and test environment to faster systems like using a runner that is for cloud environment to reduce latency.
- **Enable Fail-Fast Mechanisms:** Stop the pipeline as soon as a critical step fails.
- **Use Pre-Built Docker Images:** Avoid spending time installing tools or dependencies during the pipeline execution.

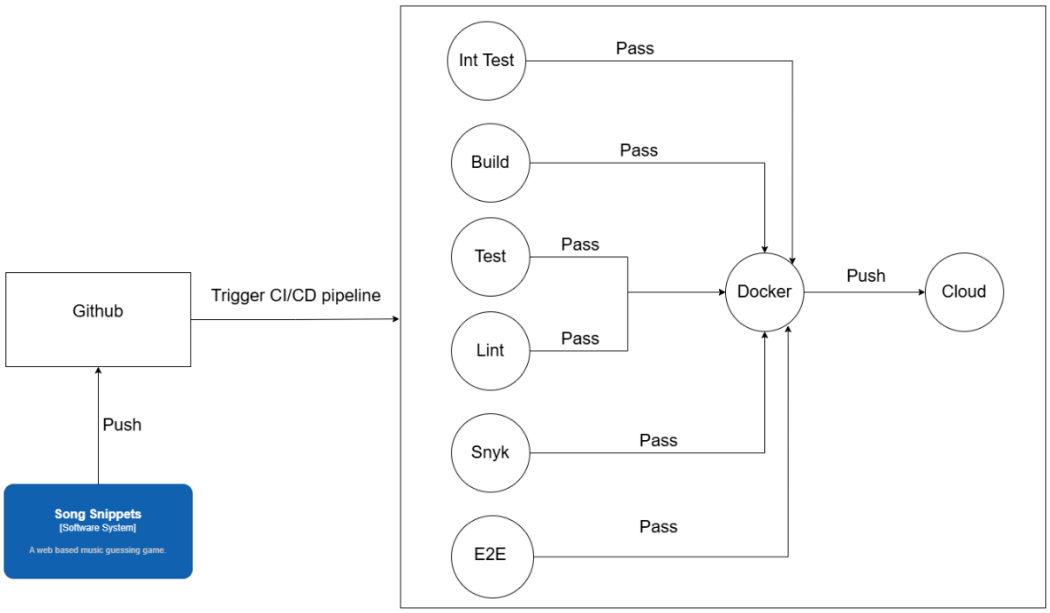
https://medium.com/@ali_hamza/from-45-minutes-to-10-minutes-how-i-made-my-ci-cd-pipeline-5x-faster-9e8bbacc6f29

https://hackernoon.com/how-to-optimize-your-cicd-pipeline-for-maximum-efficiency

Here is the second version of the CI/CD pipeline design, incorporating the optimizations.



Now the tasks are executing in parallel, it still missing some tasks like integration test and end to end test.



Here is the final version of the CI/CD pipeline design. At the time of writing, the integration tests, end-to-end tests, and automated deployment are still missing from the pipeline. These tasks will be implemented in the project before the submission deadline.

Conclusion

Each service must have its own repository and implement a CI/CD pipeline. In my case, I'm using GitHub for version control and GitHub Actions to set up the CI/CD pipeline. The pipeline for Golang follows a similar structure to pipelines for other programming languages. The setup includes:

- Build
- Test
- Lint (Golang-specific) or SonarQube (for other programming languages)
- Snyk (dependency and security scanning)
- Integration tests
- End-to-end tests
- Build Docker image
- Deploy to the cloud

4. What are the best practices for deploying Golang microservices using serverless cloud functions?

What is cloud function?

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.

https://cloud.google.com/functions/docs/concepts/overview

https://en.wikipedia.org/wiki/Serverless_computing

https://www.ibm.com/think/topics/serverless

How to deploy your microservice to cloud?

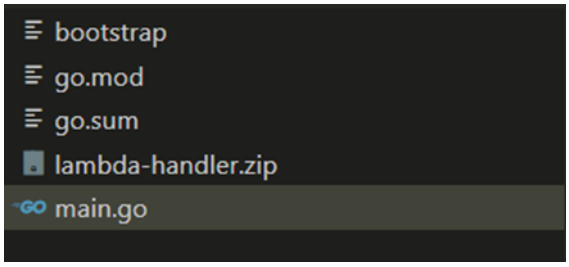
I wrote an in-depth, step-by-step guide on deploying my song service to a serverless cloud in the "Technology Research" document under the section "How to deploy manually for Golang." For the serverless cloud, I used AWS Lambda and performed the deployment manually to understand the process. Later, I plan to automate this in my CI/CD pipeline.

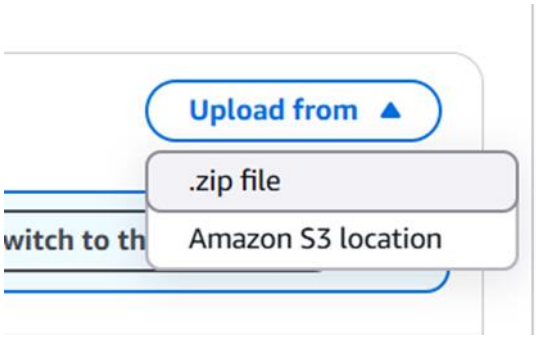
Here, I will summarize the manual deployment process. First, you need to create a Lambda function based on the specific functionality you want to deploy. In my case, it was the "Create Song" function.

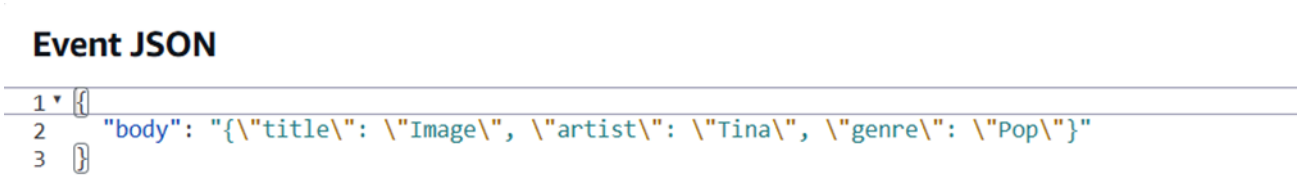
```go
func lambdaHandler(ctx context.Context, req events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
    var newSong Song
    if err := json.Unmarshal([]byte(req.Body), &newSong); err != nil || newSong.Title == "" || newSong.Artist == "" || newSong.Genre
        return events.APIGatewayProxyResponse{StatusCode: 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)
}
```

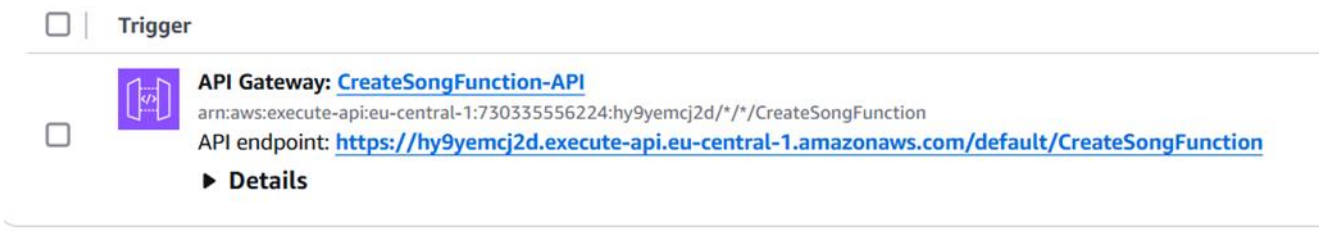
After that, you need to build the binary code and zip it. In the Lambda UI, I upload the zip file. Once the file is uploaded, you can test the function in Lambda. If the test passes, then you know it works successfully.

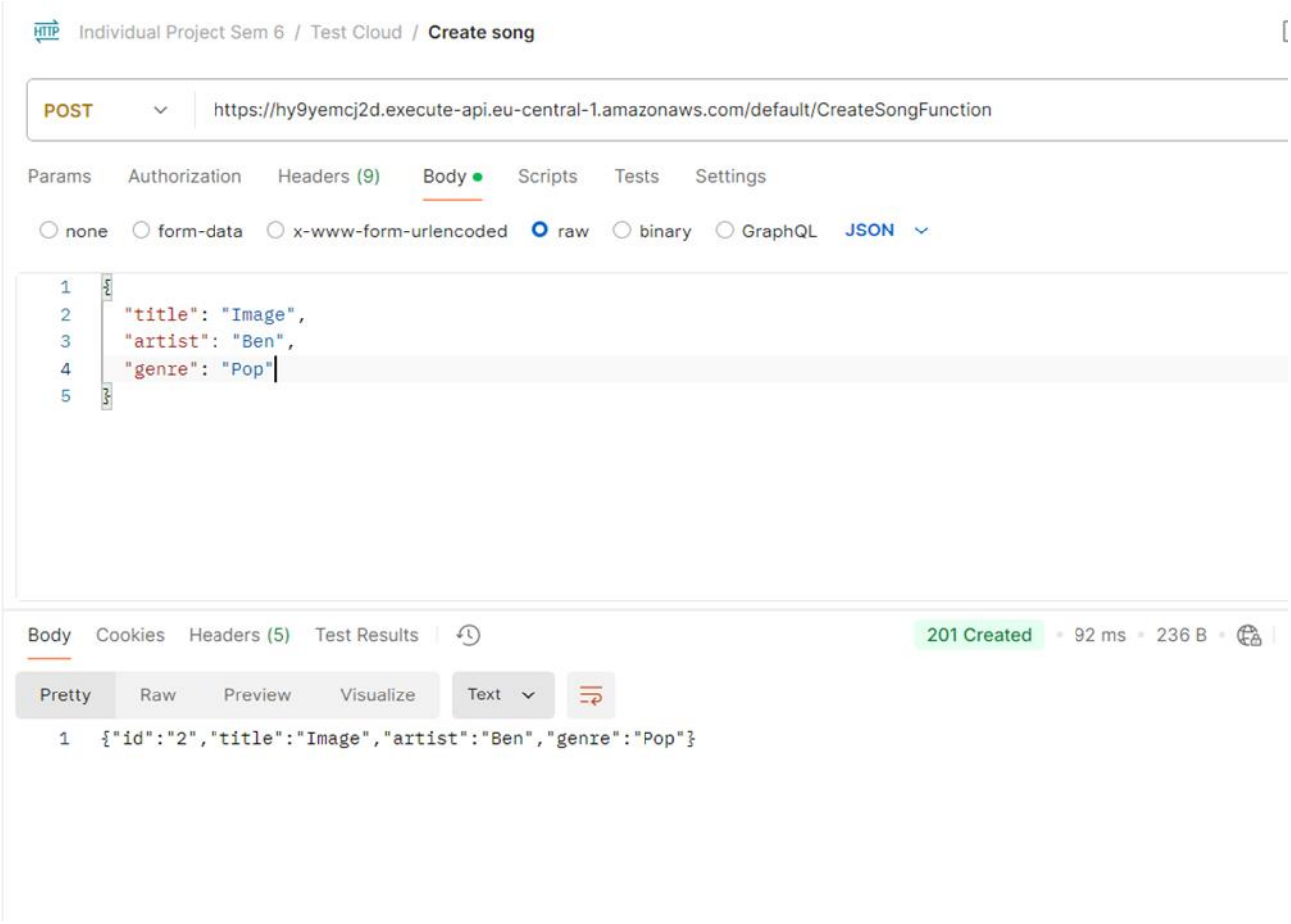






Next, I added an API Gateway to trigger the function and tested the API Gateway using Postman to ensure it was set up correctly.





What's the best practices in deploying Golang to serverless cloud?

I my case I was using AWS Lambda. Deploying Golang microservices using AWS Lambda requires a tailored approach to ensure efficiency, scalability, and cost-effectiveness. Here are best practices for this setup:

- **Optimize for Serverless Execution:** Use a lightweight Golang runtime to reduce initialization time. Avoid large dependencies by using the standard library and including only necessary packages.
- **Code Structuring:** Separate Lambda functions from the local ones. This makes local testing and cloud deployment much easier.
- **Environment Management:** Store configuration data, such as database URLs, API keys, and service-specific settings, in environment variables. Alternatively, use cloud-native tools like AWS Secrets Manager, Google Secret Manager, or Azure Key Vault to securely manage sensitive information.
- **Performance Optimization:** Keep deployment packages (e.g., ZIP files) small to speed up deployments and reduce cold start times. Use structured logging (e.g., in JSON format) to

integrate seamlessly with cloud logging services like AWS CloudWatch or Google Cloud Logging.

- **Integration with Other Services:** Use event triggers, such as HTTP requests, message queues (e.g., RabbitMQ or cloud-native alternatives), or database events to invoke functions. Consider using cloud-based databases for easier scalability.
- **CI/CD for Serverless**: Integrate CI/CD tools like GitHub Actions, AWS CodePipeline, or GitLab CI/CD for seamless deployments.
- **Monitoring and Observability:** Use cloud-native monitoring tools to track key metrics like invocation count, duration, and error rates.

https://www.liquidweb.com/blog/cloud-microservices/

https://medium.com/all-about-tech-and-techies/go-in-the-cloud-best-practices-for-cloud-based-golang-applications-41e74cc30896

Conclusion

In my case, I was using AWS Lambda, so the best practices for deploying a Golang microservice to AWS Lambda include the following:

- Use a lightweight Golang service with minimal logic, dependencies, and libraries.
- Structure your code for better maintainability.
- Store any secrets in environment variables or, alternatively, use secret management tools.
- Keep your ZIP file small and include structured logging.
- Use message queues and cloud databases.
- Integrate CI/CD for seamless deployments.
- Use monitoring tools to track performance and errors.

For other serverless clouds, the deployment process might vary slightly, but the best practices would remain the same.

5. What tools can be used to monitor the scalability of microservices?

What is monitoring?

Monitoring in a microservice architecture refers to the practice of observing, tracking, and collecting metrics and logs from various components (services) to ensure their health, performance, and overall reliability. This helps in detecting issues early, troubleshooting problems, and ensuring the system operates smoothly under varying loads.

Monitoring also plays a crucial role in scalability. Here's how you can monitor scalability:

- **Auto-Scaling Triggers:** You can monitor if your service is scaling horizontally by adding more instances, or vertically when an instance reaches its limit and triggers scaling, whether through Kubernetes or in the cloud.
- **Identifying Bottlenecks**: Monitoring helps pinpoint bottlenecks in specific services that experience high latency. This information can help you come up with solutions to improve the service or optimize scaling for that service.
- **Efficient Resource Allocation**: Monitoring provides insights into which services demand high or low resources, helping you decide which services should scale more or less.
- **Auto-Scaling in the Cloud:** By monitoring, you can adjust the threshold for auto-scaling limits to optimize resource usage.

https://swimm.io/learn/microservices/microservices-monitoring-importance-metrics-and-5-critical-best-practices#:~:text=Microservices%20monitoring%20is%20the%20practice,within%20a%20larger%20application%20architecture.

https://medium.com/cloud-native-daily/microservices-monitoring-and-observability-in-depth-d40aa0795dd3

What tools are there for monitoring microservice?

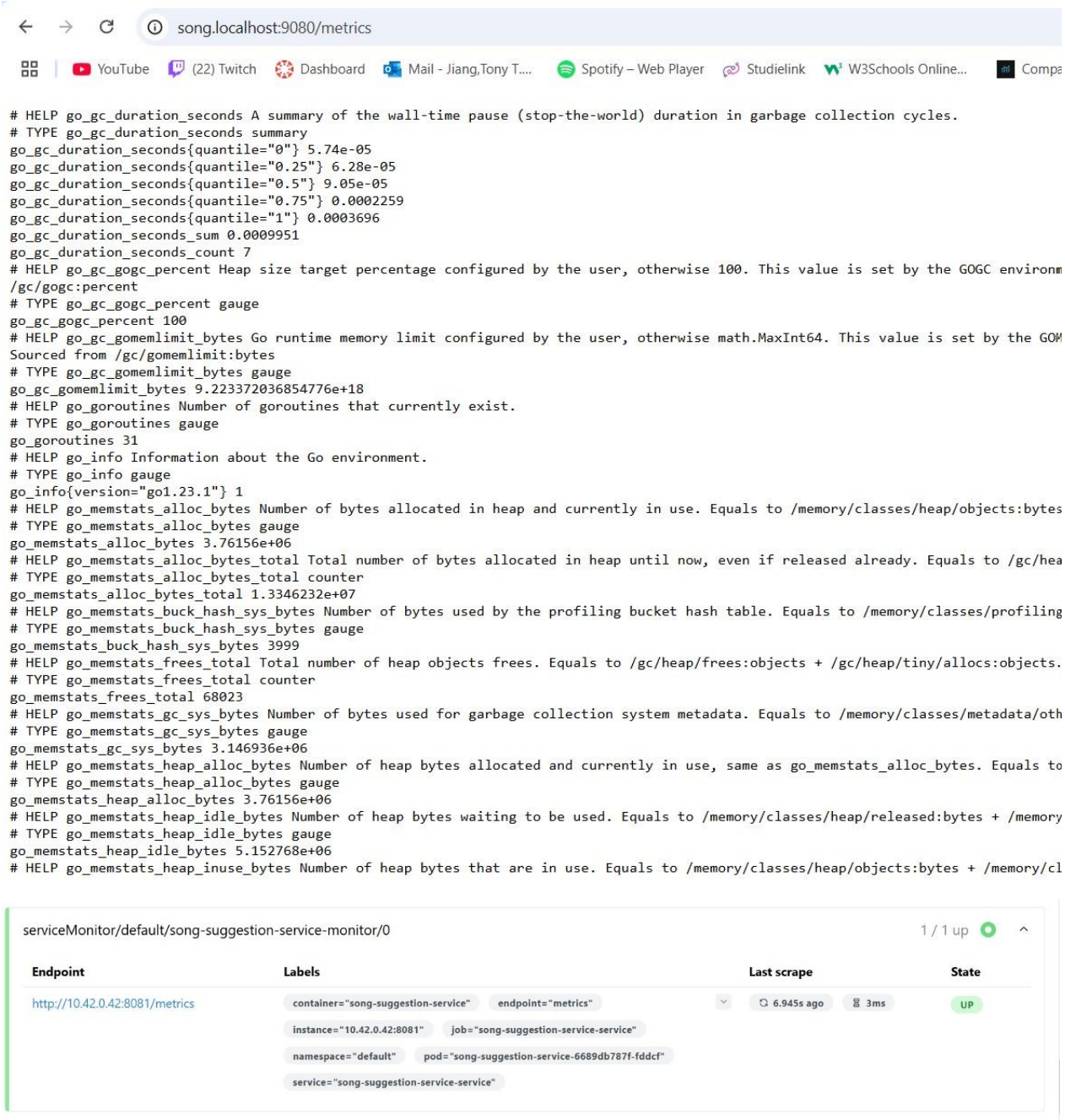
There are many tools available to monitor microservices, so I'm going to focus on the top 5 best used monitoring tools on the internet. Here they are:

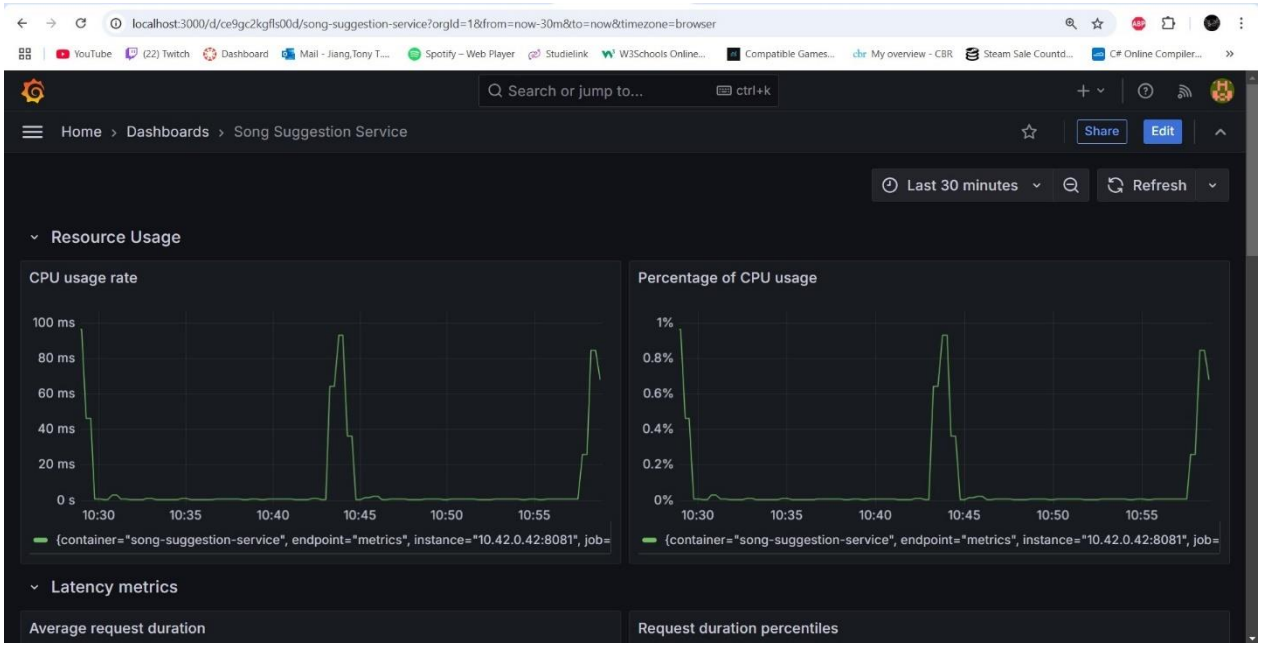
- **Prometheus:** Prometheus is a widely adopted open-source monitoring tool known for its efficiency and scalability. It uses a unique labeling system that allows for flexible querying and aggregation of metrics.
- **Grafana:** Grafana is a powerful analytics platform that complements monitoring tools like Prometheus. It provides rich features for visualizing time series data and creating custom dashboards.
- **Nagios:** Nagios is a well-established monitoring tool known for its comprehensive capabilities. It offers various plugins and integrations, making it suitable for monitoring different services and infrastructures.
- **Zabbix:** Zabbix is an open-source monitoring solution that excels in network monitoring. It offers a robust and customizable approach to monitoring microservices and their underlying infrastructure.
- **Icinga:** Icinga is an advanced monitoring tool focused on ensuring high availability and identifying performance trends. It offers a modern and intuitive user interface that simplifies the monitoring process.

https://openobserve.ai/resources/microservices-monitoring-tools-2024

https://blog.bitsrc.io/best-microservices-monitoring-tools-to-follow-in-2023-489c993b57c6

Here I am using Prometheus to monitor my song suggestion service deployed on Kubernetes (k3d). I used Prometheus to gather metrics during the load test and Grafana to visualize the monitoring data.





Build in monitoring from cloud provider

Some cloud providers have their own monitoring tools built into their deployment environments. For example, AWS Lambda has its own monitoring tool for the functions that you deploy.

Conclusion

There are many tools available for monitoring microservices, so I've decided to highlight the top 5 best tools based on the internet:

- Prometheus
- Grafana
- Nagios
- Zabbix
- Icinga

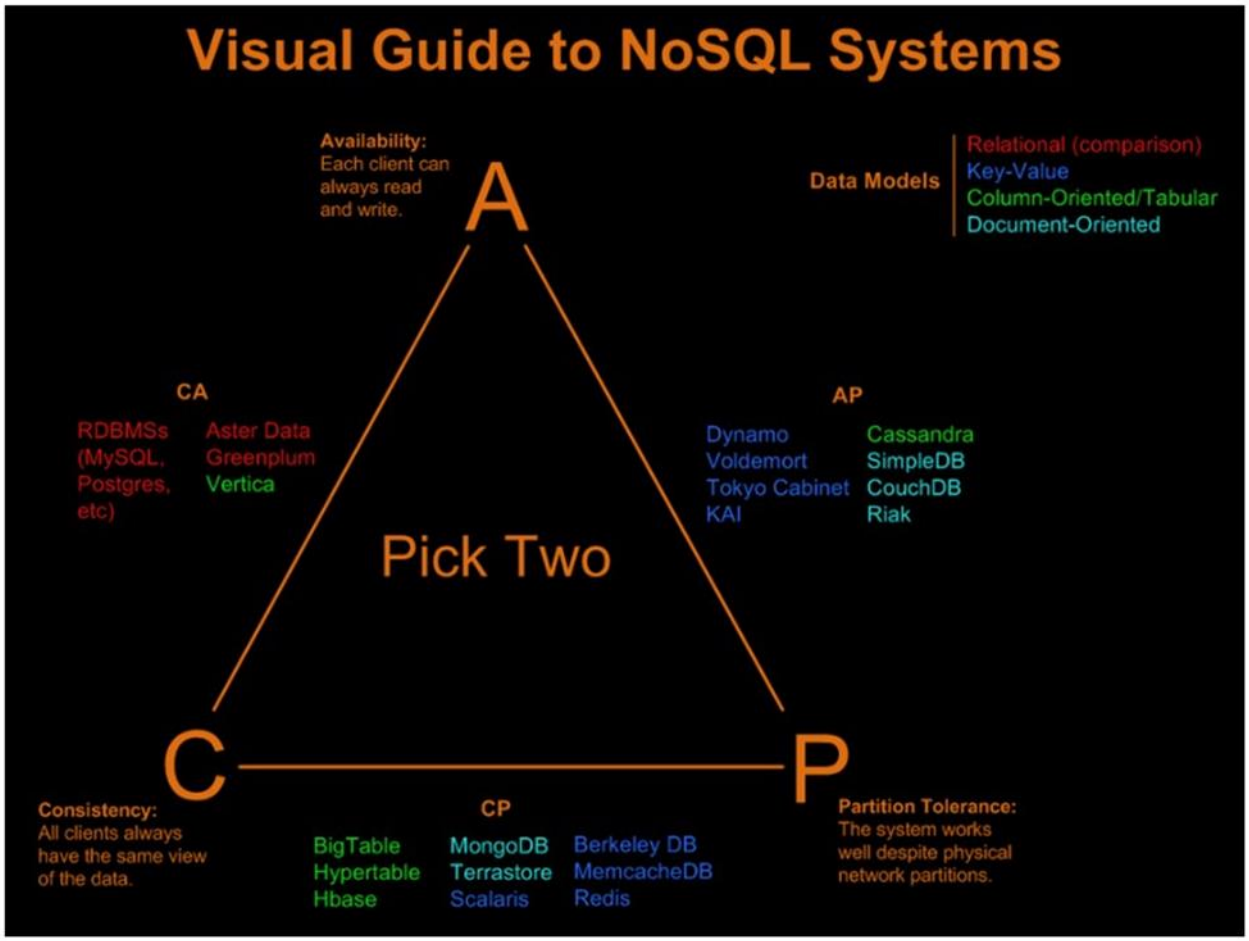
Personally, I've used Prometheus and Grafana. I use Prometheus to collect metrics and Grafana to display the visualizations of the monitoring data. I chose these tools because they are open source and it the most used.

6. How should data be managed across distributed microservices to ensure consistency, reliability, and scalability?

Firstly, you need to identify the type of database your system requires. To achieve this, I used the CAP theorem to determine the most suitable database for my system. I found this approach in my

school Canvas resources and explored it in depth in the "Database Selection (Individual)" section of my Technology Research document. Based on the analysis, I determined that my system needs to prioritize AP (Availability + Partition Tolerance) to ensure quick response times and resilience to network issues.

In my use case, users prefer faster responses over strict data consistency, as they are playing a game where immediate feedback on their answers is more valuable than perfectly consistent data. Consequently, my system will utilize an AP-focused database.



You can use the CAP theorem to determine the type of data each service needs based on its priorities.

How to ensure data consistency?

Consistency in data means that users see the updated version of the data in real time.

To ensure data consistency, you need to choose the appropriate database types for each service. For strong consistency, relational databases like PostgreSQL or MySQL are recommended. You can implement a message queue across services to ensure data consistency, using tools like Kafka or RabbitMQ.

https://dilfuruz.medium.com/data-consistency-in-microservices-architecture-5c67e0f65256

https://daily.dev/blog/10-methods-to-ensure-data-consistency-in-microservices

How to ensure data reliability?

Data reliability means ensuring that the data you use in your system is trustworthy, safe, and available whenever you need it. Even if something goes wrong, such as a system crash or a network issue, reliable data ensures it won't get lost or corrupted and will still be accessible when you try to retrieve it.

To ensure data reliability, you should store the data in multiple databases, replicate it, or implement backups to prevent data loss due to network or hardware failures. Encryption during transit is also essential. Additionally, you can use circuit breakers (e.g., Hystrix or Resilience4j) to prevent failures from cascading across microservices and affecting overall reliability.

https://milanbrankovic.medium.com/microservices-reliability-49937e11e81d

https://www.linkedin.com/advice/3/how-do-you-ensure-your-microservices-reliable

How to ensure data scalability?

Scaling data refers to the ability of a system to handle increasing amounts of data and traffic without sacrificing performance. As your system grows, you need to ensure that the data storage and processing can scale accordingly.

You can scale data through vertical or horizontal scaling. Vertical scaling means adding more resources, such as CPU or memory, to the server. Horizontal scaling means adding more instances or servers instead of increasing resources on a single server. NoSQL-based databases are ideal for this, as they are designed to scale horizontally.

https://medium.com/design-microservices-architecture-with-patterns/scaling-databases-in-microservices-architecture-with-horizontal-vertical-and-functional-data-537c5aea41d6

https://karandeepsingh.ca/post/how-to-scale-database-in-microservices/

Conclusion

Managing data across distributed microservices requires a strategic approach to ensure consistency, reliability, and scalability.

To achieve data consistency, it is important to select the appropriate database types, such as relational databases for strong consistency, and to implement message queues like Kafka or RabbitMQ for cross-service data synchronization.

For data reliability, leveraging multiple databases, replicating data, and implementing backups can help protect against data loss caused by network or hardware failures. Additionally, using encryption during transit and circuit breakers (e.g., Hystrix or Resilience4j) can enhance data reliability by preventing cascading failures across microservices.

To ensure data scalability, vertical and horizontal scaling methods should be used to handle increasing traffic and data volumes, with NoSQL databases offering effective horizontal scaling solutions.

By addressing these key areas, you can ensure that your distributed microservices architecture remains robust, efficient, and capable of handling growing demands while maintaining high performance and availability.

Conclusion

Designing a scalable web-based music guessing game using Golang within a microservice architecture requires careful planning, designing and implementation to ensure scalability, maintainability, and security. By leveraging the power of Golang's strong concurrency and the flexibility of a microservice architecture, key design principles such as deploying to Kubernetes for horizontal scaling, decoupling services, and using Docker for containerization are essential. Employing asynchronous messaging systems like Kafka or RabbitMQ ensures effective communication between services while maintaining independence.

Security is a top priority, and regular OWASP Top 10 reports should be generated and addressed in each sprint to safeguard against vulnerabilities. Additionally, each service should have its own repository, database, and a fully integrated CI/CD pipeline for continuous delivery, testing, and deployment.

When deploying to AWS Lambda or similar serverless environments, it's important to follow best practices like minimizing dependencies, structuring code for maintainability, and utilizing cloud services such as message queues and databases. Moreover, continuous monitoring using tools like Prometheus and Grafana is essential to track performance and ensure the health of the system.

Finally, ensuring data consistency, reliability, and scalability is vital to a successful microservice design. Implementing the right database types, using replication and backups, and applying horizontal scaling techniques will enable the game to scale smoothly as traffic and data increase. By adopting these principles, you can build a robust, secure, and scalable web-based music guessing game that can handle growth and provide a seamless user experience.