Backend Assignment

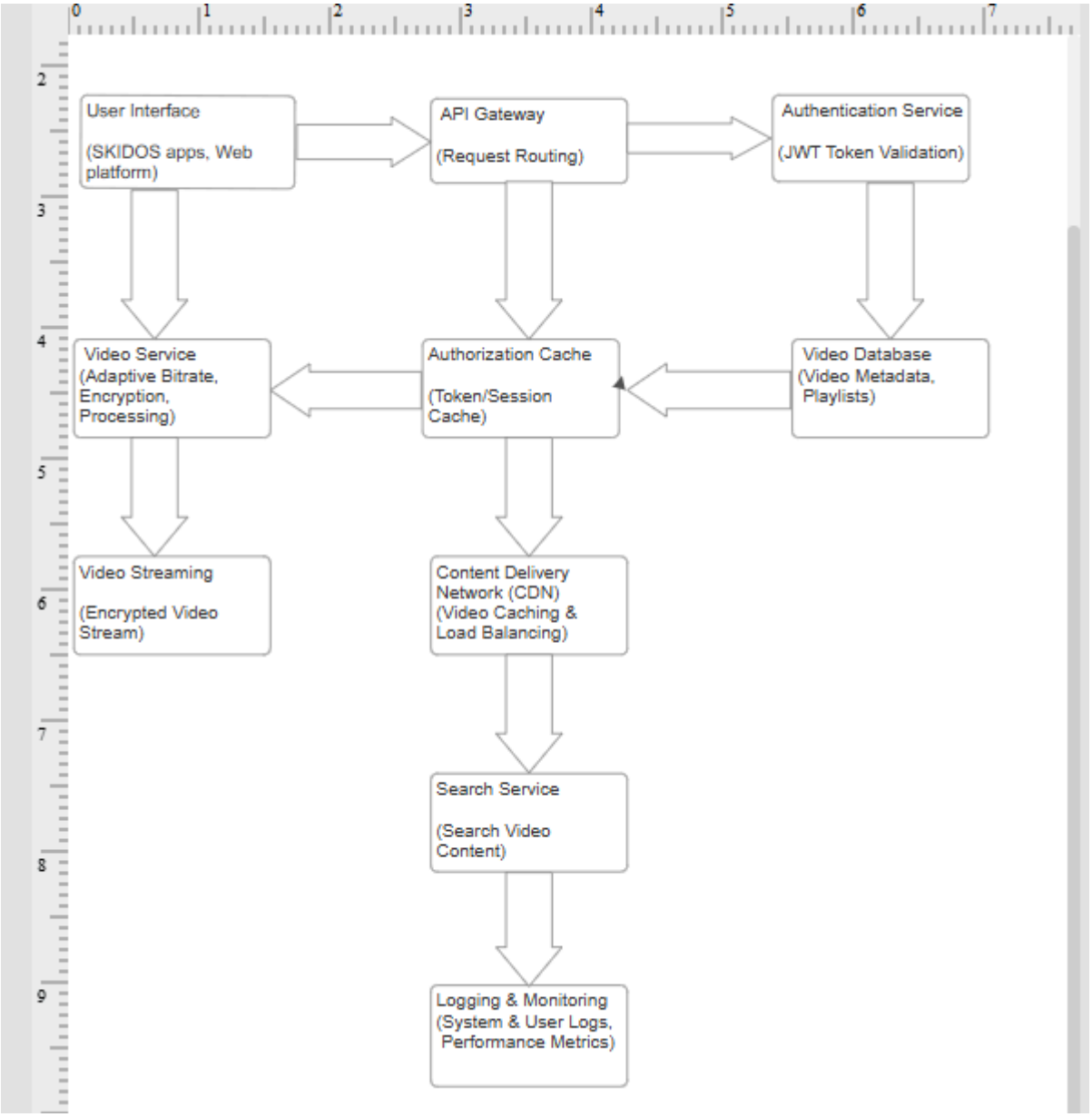
SKIDOS is continuously working on making learning a fun activity. We find videos that make
Learning is fun and exciting. Videos can help kids acquire some crucial 21st-century skills.

Assignment 1:
Create HLD and LLD for a Video on Demand backend microservice to serve the fun learning
video content through our SKIDOS apps and web platform.
Things to consider while creating microservice HLD and LLD:
1. We have a playlist of 250+ videos and continuously adding
2. Each video is of 1920x1080px
3. The average length of a video is 15 mins, and the average video size is 950 MB

Required features:
1. Only authenticated users should have access to the videos
2. Video microservice should support adaptive bitrate streaming
3. Video microservice should support encryption of video

Good to have features:
1. Video search feature
Share the HLD, LLD, and supporting documentation for the VoD backend microservice.

High-Level Design (HLD) block diagram for Video on Demand (VoD) service backend



How It Works:

- User Interface (SKIDOS apps and web platforms) sends requests to the API Gateway.
- The API Gateway routes requests to the Authentication Service for JWT validation.
- Once authenticated, the API Gateway forwards the request to the Video Service for processing and streaming.
- The Video Service uses a Content Delivery Network (CDN) for fast video delivery and stores video metadata in the Database.
- The Search Service helps with searching videos from the metadata.

Explanation of System Components

User Interface (SKIDOS apps, Web Platform)

- **Purpose**: Serves as the primary interaction point for users to access videos. This includes mobile and web applications where users initiate video requests.
- **Functionality**: Sends video requests and search queries to the API Gateway.

API Gateway

- **Purpose**: Acts as the system's central access point, directing incoming user requests to the correct services.
- **Functionality**: Routes authentication requests to the Authentication Service and forwards validated requests to the Video Service.

Authentication Service

- **Purpose**: Manages user validation by verifying login credentials and issuing secure JWT (JSON Web Tokens) for accessing videos.
- **Functionality**: Checks user tokens for authentication and sends authenticated requests back to the API Gateway.
- **Additional Component**: An Authorization Cache temporarily stores tokens and session details for quick access.

Video Service

- **Purpose**: Handles core video-related operations such as decryption, adaptive bitrate streaming, and delivering content.
- **Functionality**: Processes video requests and delivers content either directly or via a Content Delivery Network (CDN).

- **Interactions**: Connects with the database to retrieve video metadata, user preferences, and playback history.

Video Streaming

- **Purpose**: Ensures smooth video delivery to users.
- **Functionality**: Streams content processed by the Video Service, with optional encryption. Videos are delivered either from the CDN or directly from the Video Service.

Content Delivery Network (CDN)

- **Purpose**: Improves video accessibility by caching content closer to the user's location. This reduces latency and balances server load.
- **Functionality**: Frequently accessed videos are cached for faster delivery, easing the load on the Video Service.

Video Database

- **Purpose**: Stores video-related information, such as metadata, titles, descriptions, tags, and user playlists.
- **Functionality**: Supplies the Video Service with metadata and manages user-specific data like watch histories and playlists.

Search Service

- **Purpose**: Enables users to locate videos using criteria like titles, tags, or categories.
- **Functionality**: Fetches video metadata from the database to facilitate searches.

Logging and Monitoring

- **Purpose**: Tracks system activities, user interactions, and performance metrics.
- **Functionality**: Collects performance data and logs errors to provide insights into system health and usage patterns.

Data Flow and Communication

1. **User Request Process**: Requests originate from the user interface (mobile or web) and are forwarded to the API Gateway. The API Gateway handles authentication by communicating with the Authentication Service.
2. **Video Handling**: Once authenticated, the API Gateway routes requests to the Video Service. The service processes the request and retrieves metadata from the database before delivering the content directly or via the CDN.

3. **Search and Monitoring**: Search queries are processed by the Search Service using the database, while Logging and Monitoring components track and analyze system interactions for efficiency.

Key Design Considerations

- **Scalability**: Components like the Video Service and CDN should support horizontal scaling to handle increased demand.
- **Security**: Strong encryption and robust JWT management ensure secure video delivery.
- **Performance**: CDNs minimize buffering and improve load times, enhancing the user experience.
- **Resilience**: Implementing redundancy and failover mechanisms ensures high availability.

Low-Level Design (LLD) implementation of the Video on Demand (VoD) :-

1. Authentication Service

This component checks if a user is authenticated using JWT tokens.

```
package auth


import (
    "github.com/dgrijalva/jwt-go"
    "time"
)


const privateKey = "mySecret"


// ValidateUserToken checks the validity of the provided JWT token.
func ValidateUserToken(userToken string) (bool, error) {
    _, err := jwt.Parse(userToken, func(token *jwt.Token) (interface{}, error) {
        return []byte(privateKey), nil
    })
    if err != nil {
        return false, err
    }
```

```
    return true, nil
}

// GenerateUserToken creates a JWT token for an authenticated user.
func GenerateUserToken(username string) (string, error) {
    claims := jwt.MapClaims{
        "user": username,
        "exp":  time.Now().Add(time.Hour * 24).Unix(),
    }
    jwtToken := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return jwtToken.SignedString([]byte(privateKey))
}
```

2. Video Processing and Streaming

This service handles video processing like encryption and bitrate adaptation.

```
package video

import (
    "fmt"
    "time"
)

type VideoFile struct {
    ID          int
    Title       string
    Description string
    Quality     string
    IsEncrypted bool
    FilePath    string
}

type VideoStreamService struct {
    VideoRepo  *VideoRepository
    CacheSystem *CacheService
}
```

```
func (vs *VideoStreamService) RetrieveVideoDetails(videoID int) (*VideoFile, error) {
    return vs.VideoRepo.FetchVideoByID(videoID)
}

func (vs *VideoStreamService) ProcessVideoForStreaming(videoID int) (*VideoFile, error) {
    video, err := vs.RetrieveVideoDetails(videoID)
    if err != nil {
        return nil, err
    }

    // Simulate video processing: encryption & bitrate adjustment
    video.IsEncrypted = true
    video.Quality = "adaptive"

    // Save the processed video to CDN (Cache)
    err = vs.CacheSystem.SaveVideoToCache(video)
    if err != nil {
        return nil, err
    }

    return video, nil
}

func (vs *VideoStreamService) StreamVideoContent(videoID int) (*VideoFile, error) {
    video, err := vs.ProcessVideoForStreaming(videoID)
    if err != nil {
        return nil, err
    }

    fmt.Printf("Streaming video: %s\n", video.Title)
    return video, nil
}
```

3. Cache Service (CDN)

This service handles video caching for efficient delivery.

```
package cache

import (
    "errors"
    "fmt"
)

type VideoFile struct {
    ID       int
    FilePath string
}

type CacheService struct {
    VideoCache map[int]*VideoFile
}

// NewCacheService initializes the cache system.
func NewCacheService() *CacheService {
    return &CacheService{
        VideoCache: make(map[int]*VideoFile),
    }
}

// SaveVideoToCache saves a video to the cache.
func (cs *CacheService) SaveVideoToCache(video *VideoFile) error {
    if video == nil {
        return errors.New("invalid video")
    }
    cs.VideoCache[video.ID] = video
    return nil
}

// GetVideoFromCache retrieves a video from the cache by ID.
func (cs *CacheService) GetVideoFromCache(videoID int) (*VideoFile, error) {
    video, found := cs.VideoCache[videoID]
    if !found {
```

```
        return nil, fmt.Errorf("video not found in cache: %d", videoID)
    }
    return video, nil
}
```

4. Video Repository (Database)

This component simulates the database interactions for video storage and retrieval.

```
package repository

type VideoFile struct {
    ID          int
    Title       string
    Description string
    FilePath    string
}

type VideoRepository struct {
    VideoStore map[int]*VideoFile
}

// NewVideoRepository initializes a new repository for video storage.
func NewVideoRepository() *VideoRepository {
    return &VideoRepository{
        VideoStore: make(map[int]*VideoFile),
    }
}

// StoreVideo adds a video to the repository.
func (vr *VideoRepository) StoreVideo(video *VideoFile) {
    vr.VideoStore[video.ID] = video
}

// FetchVideoByID retrieves video details from the repository.
func (vr *VideoRepository) FetchVideoByID(videoID int) (*VideoFile, error) {
    video, exists := vr.VideoStore[videoID]
```

```
    if !exists {
        return nil, fmt.Errorf("video not found: %d", videoID)
    }
    return video, nil
}
```

5. Video Search Service

The search service helps users search for videos by title or description.

```
package search

import "fmt"

type VideoFile struct {
    ID          int
    Title       string
    Description string
}

type SearchService struct {
    VideoRepo *VideoRepository
}

func NewSearchService(videoRepo *VideoRepository) *SearchService {
    return &SearchService{VideoRepo: videoRepo}
}

// FindVideos searches for videos matching the query in title or description.
func (ss *SearchService) FindVideos(query string) ([]*VideoFile, error) {
    var foundVideos []*VideoFile
    for _, video := range ss.VideoRepo.VideoStore {
        if contains(video.Title, query) || contains(video.Description, query) {
            foundVideos = append(foundVideos, video)
        }
    }
    return foundVideos, nil
```

```
}

// contains checks if a substring exists in a string.
func contains(source, target string) bool {
    return fmt.Sprintf("%s", source) == target
}
```

6. API Gateway

This component coordinates requests, ensuring video streaming and search functionality.

```
package api

import (
    "fmt"
    "video-on-demand/auth"
    "video-on-demand/video"
    "video-on-demand/search"
)

type Gateway struct {
    AuthSvc     *auth.AuthenticationService
    VideoSvc    *video.VideoStreamService
    SearchSvc   *search.SearchService
}

func NewGateway(authSvc *auth.AuthenticationService, videoSvc *video.VideoStreamService, searchSvc
*search.SearchService) *Gateway {
    return &Gateway{
        AuthSvc:   authSvc,
        VideoSvc:  videoSvc,
        SearchSvc: searchSvc,
    }
}

// ProcessRequest handles video streaming requests.
```

```
func (g *Gateway) ProcessRequest(token string, videoID int) (*video.VideoFile, error) {
    valid, err := g.AuthSvc.ValidateUserToken(token)
    if err != nil || !valid {
        return nil, fmt.Errorf("authentication failed: %v", err)
    }

    return g.VideoSvc.StreamVideoContent(videoID)
}

// ProcessSearchRequest handles video search requests.
func (g *Gateway) ProcessSearchRequest(query string) ([]*video.VideoFile, error) {
    return g.SearchSvc.FindVideos(query)
}
```

Main Program

This is the main entry point to simulate how all the services interact.

```
package main

import (
    "fmt"
    "video-on-demand/auth"
    "video-on-demand/video"
    "video-on-demand/repository"
    "video-on-demand/search"
    "video-on-demand/cache"
    "video-on-demand/api"
)

func main() {
    // Initialize all services
    videoRepo := repository.NewVideoRepository()
    cacheSvc := cache.NewCacheService()
    authSvc := auth.NewAuthenticationService()
    videoSvc := video.NewVideoStreamService(videoRepo, cacheSvc)
    searchSvc := search.NewSearchService(videoRepo)
```

```
    apiGateway := api.NewGateway(authSvc, videoSvc, searchSvc)

    // Add videos to repository
    videoRepo.StoreVideo(&repository.VideoFile{ID: 1, Title: "Learning Made Fun", Description: "A fun video to teach kids."})
    videoRepo.StoreVideo(&repository.VideoFile{ID: 2, Title: "Adventure Time", Description: "An exciting adventure video for kids."})

    // Search videos
    foundVideos, _ := apiGateway.ProcessSearchRequest("Learning")
    fmt.Printf("Found %d video(s) matching the search criteria\n", len(foundVideos))

    // Authenticate and stream a video
    token, _ := authSvc.GenerateUserToken("johnDoe")
    video, err := apiGateway.ProcessRequest(token, 1)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("Now Streaming:", video.Title)
    }
}
```

Explanation:

1. Authentication Service validates users via JWT tokens.
2. Video Service processes the videos, including encryption and adaptive bitrate streaming.

3. Content Delivery Network (CDN) caches videos for fast delivery.
4. Video Database stores and manages video metadata.
5. Video Search Service allows searching for videos.
6. API Gateway routes requests and ensures that authentication is validated before allowing access to video content.

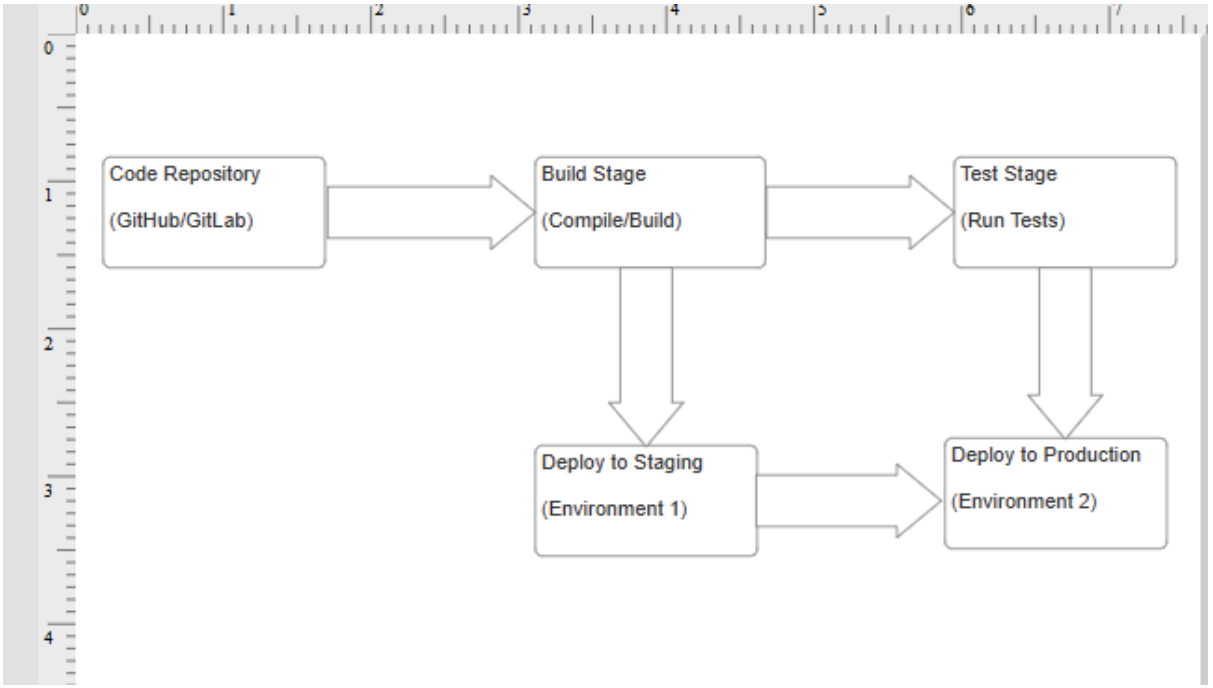
Assignment 2:

Create a CI/CD pipeline that automatically integrates, builds and tests code changes with a
repository. It should automatically deliver the code changes to the staging and production
environment.

1. Share the HLD of the CI/CD pipeline and supporting documentation.

The CI/CD pipeline is an automated process that includes:

Integration Stage: Automatically integrates code changes from the repository (usually Git).
Build Stage: Automatically builds the code, compiling and packaging it into an executable or a deployable artefact.
Test Stage: Automatically runs tests to ensure that the code is correct and doesn't break functionality.
Deploy to Staging: Once the tests are passed, deploy the code to a staging environment.
Deploy to Production: After successful deployment and verification in staging, deploy the code to the production environment.



Explanation of the HLD:

1. Code Repository: Developers push code changes to the repository (e.g., GitHub, GitLab). The repository contains the source code and configurations.
2. Build Stage: The CI/CD tool detects changes in the repository, triggers the build process, and compiles the application.
3. Test Stage: Once the build is complete, automated tests are executed to validate the correctness of the code.
4. Deploy to Staging: If tests pass, the code is automatically deployed to the staging environment for further validation.
5. Deploy to Production: After successful validation in the staging environment, the code is deployed to the production environment.

Pipeline in Golang:

Here's how you might create a simple Golang-based application in the CI/CD pipeline:

Example: GitHub Actions Workflow (CI/CD Pipeline) with Golang Integration

```
name: Go CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Go
      uses: actions/setup-go@v3
      with:
        go-version: '1.18'

    - name: Install dependencies
      run: |
        go mod tidy

    - name: Run tests
      run: |
        go test ./...

    - name: Build application
      run: |
        go build -o myapp .
```

```
- name: Build Docker image
  run: |
    docker build -t myapp .

- name: Push Docker image to DockerHub
  run: |
    docker login -u ${{ secrets.DOCKER_USERNAME }} -p ${{ secrets.DOCKER_PASSWORD }}
    docker tag myapp:latest mydockerusername/myapp:latest
    docker push mydockerusername/myapp:latest

- name: Deploy to Staging Environment
  run: |
    kubectl apply -f kubernetes/staging-deployment.yaml

- name: Deploy to Production Environment
  run: |
    kubectl apply -f kubernetes/production-deployment.yaml
```

Explanation of Pipeline Stages in Golang:

1. Checkout code: This step checks out the latest code from the repository.
2. Set up Go environment: Installs the specific version of Go (e.g., Go 1.18) required for the build.
3. Install dependencies: Runs `go mod tidy` to ensure that all dependencies are available.
4. Run tests: Executes the tests to ensure that no code changes break the functionality.
5. Build application: Uses `go build` to compile the Golang application into an executable or binary.
6. Build Docker image: Builds a Docker image for the application if it's going to be containerized for deployment.
7. Push Docker image: Pushes the Docker image to DockerHub (or another container registry).

8. Deploy to Staging: Deploys the application to a staging environment using Kubernetes (kubectl).
9. Deploy to Production: If everything works well in staging, deploys the code to the production environment.