Cloud Nine

Zaki Baker Javon, David Carangan, Jordan Finch, Jack Goehringer, Jeffrey Kjelstrom
West Chester University of Pennsylvania

I. INTRODUCTION

Cloud Nine is a lightweight, container-oriented video-streaming platform designed for modern cloud environments. It combines a Flask backend, Kubernetes-orchestrated microservices, and an HLS/FFmpeg streaming pipeline to deliver a flexible, scalable video experience. The goal is to demonstrate how a cloud-native media service can be deployed using simple, modular components. The platform emphasizes portability, resilience, and maintainability while remaining approachable for developers and students learning Kubernetes. Through the power of Kubernetes, our project can be quickly and reliably deployed whether it be a personal device or a server.

II. ARCHITECTURE

To put it simply, the architecture of this project consists of several different pods, each with their own specialized functionality. This design is used to create a simple, yet complex structure that can change based on the needs of the system as well as adapt to any changes that may occur. The adaptability of the system allows for a higher level of reliability when compared to running the system with only containers. This section will outline the overall purposes of each deployment, the connections they have to one another, and the reliability techniques used.

The first deployment that a user will interact with is the backend. This service provides the majority of the network's outside-facing API endpoints. This includes serving HTML files (which contain the CSS and JS), and providing endpoints for retrieving and updating video metadata. Each of the HTML files are served at a single word endpoint (e.g. /videos) to allow for easy access to each of the pages. To avoid stressing out the streaming services, we decided to have the backend deployment also handle the database entries for videos. Video data like the title and description can both be accessed and changed through endpoints created by this deployment. In the event that an entry is submitted that uses the same video file as one that already exists, it will simply replace the existing one.

To store the video details, we created a database pod using PostgreSQL. We opted for a SQL based database simply because the data that is stored is perfect for this type of format. The database has one table titled videos that stores the name of the video, the description of the video, the name of the video file, and a sequentially generated id for each video entry. All of the columns are marked as "NOT NULL" to avoid having any entries that do not contain all of the information for a video.

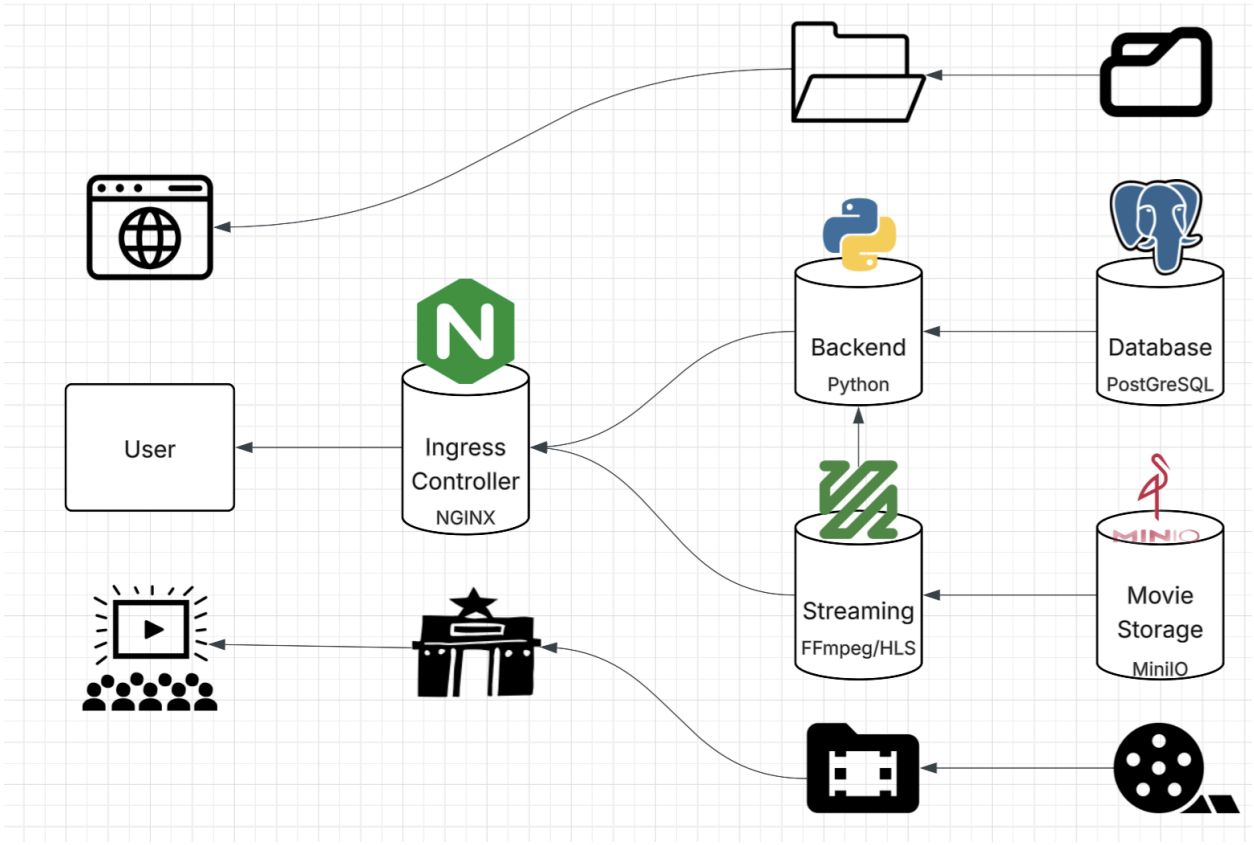
The next, and maybe most important, deployment is for streaming. Cloud Nine uses HTTP Live Streaming (HLS), which involves serving a video in smaller video files that can be more quickly downloaded. The process begins when a client requests the video manifest, which outlines the order and names of all of the video files. After that, the client can simply request each video file independently via the same URL as the manifest. The nice part about HLS is that the most popular browsers like Google's Chrome and Apple's Safari natively support it. This means that our application only has to provide the URL to the manifest file via an HTML video tag and the browser will handle the rest (given that HLS is set up correctly on the backend). For our implementation of streaming, the deployment contains two containers, one for transcoding and the other for handling HLS. When a user uploads a video to the system, it gets routed to the transcoding service. Here, the video is put into FFmpeg to be divided into multiple, smaller video files. Afterwards, the video files and the manifest created by FFmpeg are moved to the video storage to be later accessed by the HLS service. When a video file or manifest is requested by the client, the HLS server retrieves the file from the video storage and serves it to the user. The use of HLS allows for less computational strain on both the client and the server since requests only need to be made every once and a while. Both the transcoding and HLS containers use a Flask server at the base.

For storing the video files, we decided to use a deployment of MinIO. We decided to use MinIO over just storing the files in directories because of MinIO's support of Amazon's S3 and its built-in web console. The use of S3 is perfect for storing video files since it is designed to store lots of files (objects) in separate sections (buckets). Our server uses a single bucket named videos that contains buckets for each individual video. The buckets contain both the video files as well as the manifest file. MinIO's S3 API provides a simple and effective way for our streaming deployment to access the video information. Behind the scenes, when a client requests a video file the URL looks like this: "[ip address]:[port]/stream/[video name]/[file]". Since the video name is included in the URL, this makes it easy to get the file from the correct bucket since the bucket is named the same as the video.

The last, but most certainly not least, deployment is the ingress controller. This deployment handles all incoming and outgoing traffic and routes them to the correct services. The ingress controller makes it so that the entire system appears to run on a single port and all traffic comes from a single source. This makes the networking side on the front end much easier to manage. Our ingress controller is implemented using NGINX and routes all of the traffic

based on the first section of the URL path (e.g. “/upload” goes to the transcoding service and “/videos” goes to the backend).

Below is a simplified diagram that shows the architecture of Cloud Nine:



III. KUBERNETES IMPLEMENTATION

The implementation of Kubernetes into this project was pretty straight forward. Our original design for Cloud Nine had already involved several Docker containers that connected via a Docker network. Using Kubernetes involves defining how many replicas there will be, what happens when a pod fails, and how storage will be handled. These three things, along with the structure of each deployment is what this section will be outlining.

The majority of the deployments use multiple replicas that allows for higher throughput and reliability. The only deployment that does not have multiple replicas is PostgreSQL. This is because natively, PostgreSQL gets very complicated when you want to replicate and have persistent storage. Building off of that, both the PostgreSQL and the MinIO storage deployments use persistent storage so that data is not lost when a pod fails or is stopped. This is the most important part since our entire project is essentially based on storing a lot of data. We did test the storage just to make sure it works. We deleted the pods, which were restarted, albeit a few seconds later, by the deployment and they were able to access the storage once again as if they never failed. To add on, the MinIO deployment uses a sidecar pod that quickly adds the videos bucket right after the MinIO server initializes.

Since each of the services are made as deployments, each of the pods are automatically designed to be able to restart in a moment's notice. This process involves Kubernetes almost instantly noticing that a pod failed, then starting a new copy of that pod. After testing our implementation of the deployments, we did find that Kubernetes was able to quickly create a new pod upon failure or deletion. This means that we can ensure that there will always be the designated number of pods running for each deployment. Scalability is also a strong suit with this design since the number of replicas can be changed depending on the admin’s needs. Each replica of the pods will behave the same way and function together to run a smooth and seamless system.

Below is an organized list of all of the deployments with their Kubernetes details:

- Ingress Controller
    - Description: Redirects traffic to either the backend or streaming via the first path in the URL. The controller also defines which port the webpage will be served on, the maximum size of a request, timeout settings, and provides observability and leader election.
    - Service type: NodePort
    - Containers: NGINX
    - Connections:
        - Backend:
            - Port: 80
            - Type: HTTP
        - Streaming (FFmpeg):
            - Port: 8080
            - Type: HTTP
        - Streaming (HLS):
            - Port: 8081
            - Type: HTTP
    - Number of replicas: 2
    - Exposed Ports:
        - 31000: For HTTP traffic in and out of the network
    - Additional Notes:
        - All traffic going to “/upload-video” goes to the transcoding service
        - All traffic going to “/stream” goes to the HLS service
        - All other traffic is routed to the backend service
- Backend
    - Description: Provides API endpoint for both retrieving static files as well as endpoints for anything other than streaming
    - Service type: CluserIP
    - Containers: Flask Web Framework
    - Connections:
        - Ingress:
            - Port: 80
            - Type: HTTP
        - Database: PostgreSQL
            - Port: 5433
            - Type: TCP
    - Exposed Ports:
        - 80: Flask server
            - Forwarded: 3000
    - Number of replicas: 2
- Database
    - Description: Uses a SQL style database to store information
    - Service type: ClusterIP
    - Containers: PostGreSQL
    - Connections:
        - Backend
            - Port: 5433
            - Type: TCP
    - Exposed Ports:
        - 5433: PostGreSQL server
            - Forwarded: 5432
    - Number of replicas: 1
    - Additional notes:
        - Uses a PersistentVolume
- Streaming
    - Description: Provides endpoints for both uploading and streaming videos. Files uploaded are transcoded and stored in the storage deployment.

    - Service type: ClusterIP
    - Containers:
        - FFmpeg with Flask
        - Flask Web Server
            - Handles HLS
    - Connections: Storage, Backend, Ingress
        - Storage:
            - Port: 9000
            - Type: S3 API (HTTP-based)
        - Ingress:
            - Port: 80
            - Type: HTTP
    - Exposed Ports:
        - 8080: FFmpeg server
        - 8081: HLS server
    - Number of replicas: 3
- Storage
    - Description: Handles persistent storage of the transcoded video
    - Service type: ClusterIP
    - Containers: MinIO
    - Connections:
        - Streaming:
            - Port: 9000
            - Type: S3 API (HTTP-based)
    - Exposed Ports:
        - 9000: MinIO S3 API
        - 9001: MinIO Console Webpage
    - Number of replicas: 2
    - Additional Notes:
        - Uses a PersistentVolume

IV. KNOWN LIMITATIONS AND FUTURE WORK

As this project was being developed, there were some clear limitations we had realized and some areas of improvement we noticed we could work on. First off, file based storage is not suitable for high volume traffic, so we would need to figure out a way to handle many users at once. Also, FFmpeg processing is synchronous, meaning under heavy loads it will cause a bottleneck. We'd need to either implement another way to transcode the videos or scale up that amount of pods implementing FFmpeg to suit the amount of users, which would not be highly feasible. Our frontend is also static, and we could improve it using modern frameworks like React or Vue. Also, with our current setup, we don't have a proper form of centralized logging or a metrics dashboard. Unfortunately, the HLS output is not cached or CDN-accelerated. This could lead to issues like buffering, high latency, server overloads, and a lack of protection against DDoS attacks. Also, many browsers do not natively support HLS.

Some things we have planned for the future include implementing autoscaling based on CPU/ video-processing load. We have also planned on adding caching layers. We could also containerize and modernize the frontend and add observability and user profiles, live streaming, and search features.

V. CONCLUSION

It is amazing how simple, yet complex a system can be with Kubernetes. With our project, Cloud Nine, we have shown that it is possible to create a streaming platform that anyone can deploy and support seamless streaming to most browsers. Cloud Nine also has great reliability and potential for scaling through the use of deployments in Kubernetes. In this document we have outlined the work we have put into creating Cloud Nine and implementing it into a Kubernetes environment. Even though there are a few things we could add to make this project even better, we believe we have created an amazing system that is reliable, streamlined, and efficient.