**Midpoint Evaluation Report**

**Kubernetes-Based Microservices Deployment for Python Video-to-Audio Converter**

**Group** 30

**Team Members**: Sumit Karki, Bhuwan Upadhyaya, Yukesh Shrestha

**Date**: Midpoint Submission

---

**1. Working Prototype Demonstration**

**1.1 Prototype Status**

We have successfully implemented a **\*\*working prototype\*\*** of the microservices system that demonstrates the complete video-to-audio conversion workflow. The prototype is fully functional and ready for deployment, though currently configured for local development using Docker Compose.

**1.2 Implemented Components**

**#Microservices Architecture**

- **Upload Service** (FastAPI): Fully functional REST API that accepts video uploads

- **Converter Service**: Worker service that processes videos using FFmpeg

- **Storage Service** (FastAPI): Service for downloading converted audio files

**Message Queue Integration**

- RabbitMQ integration complete

- Asynchronous job queuing and processing

- Persistent message queues for fault tolerance

**Database Integration**

- **MongoDB**: Stores file metadata (job\_id, status, file paths, timestamps)

- **PostgreSQL**: Stores structured job records and request logs

- Both databases fully integrated and tested

**Containerization**

- Dockerfiles created for all three microservices

- Docker Compose configuration for local development

- Services can be built and run locally

**Kubernetes Manifests**

- Complete Kubernetes deployment configurations

- ConfigMaps for application configuration

- Secrets management structure

- Horizontal Pod Autoscalers (HPA) configured

- Persistent Volume Claims for data storage

- Ready for AWS EKS deployment

**1.3 How to Test the Prototype**

1. **\*Start Services Locally\***:

```bash

cd video-to-audio

docker-compose up -d

```

2. **\*Upload a Vide\***:

```bash

curl -X POST "http://localhost:8000/upload" \

-F "file=@sample\_videos/sample1.mp4" \

-F "user\_id=test-user"

```

3. **\*Check Job Status\***:

```bash

curl "http://localhost:8000/job/{job\_id}"

```

4. **\*Download Audio\***:

```bash

curl "http://localhost:8001/download/{job\_id}" -o output.mp3

```

**1.4 Demonstration Evidence**

- **\*API Endpoints\***: Functional REST APIs with health checks

- **\*Database Operations\***: Metadata stored in MongoDB, logs in PostgreSQL

- **\*Message Processing\***: Jobs queued and processed asynchronously

- **\*File Conversion\***: FFmpeg successfully converts videos to audio

- **\*Error Handling\***: Graceful error handling implemented

---

**2. Cloud Architecture and Technologies**

**2.1 Architecture Overview**

The system follows a **\*\*microservices architecture pattern\*\*** with clear separation of concerns:

```

Client → Upload Service → RabbitMQ → Converter Worker → Storage Service

↓ ↓ ↓

MongoDB MongoDB MongoDB

PostgreSQL PostgreSQL PostgreSQL

```

**2.2 Technologies Used**

**Application Layer**

- **\*\*Python 3.11\*\***: Core programming language

- **\*\*FastAPI\*\***: REST API framework for Upload and Storage services

- **\*\*FFmpeg\*\***: Video-to-audio conversion library

- **\*\*pika\*\***: RabbitMQ client library

- **\*\*pymongo\*\***: MongoDB driver

- **\*\*psycopg2\*\***: PostgreSQL adapter

**# Infrastructure Layer**

- **\*\*Docker\*\***: Containerization platform

- **\*\*Docker Compose\*\***: Local orchestration

- **\*\*Kubernetes\*\***: Container orchestration (manifests ready)

- **\*\*RabbitMQ\*\***: Message broker for asynchronous processing

- **\*\*MongoDB\*\***: NoSQL document database

- **\*\*PostgreSQL\*\***: Relational database

**DevOps Technologies (Prepared for)**

- **\*\*AWS EKS\*\***: Kubernetes service (for deployment)

- **\*\*Helm\*\***: Kubernetes package manager (can be added)

- **\*\*Prometheus\*\***: Monitoring (planned)

- **\*\*Grafana\*\***: Visualization (planned)

- **\*\*GitHub Actions\*\***: CI/CD (planned)

**2.3 Design Patterns Implemented**

1. **\*\*Microservices Pattern\*\***: Separate services for each function

2. **\*\*Message Queue Pattern\*\***: Decoupled async processing via RabbitMQ

3. **\*\*Worker Pattern\*\***: Multiple converter workers for parallel processing

4. **\*\*API Gateway Pattern\*\***: RESTful APIs for service access

5. **\*\*Database per Service\*\***: MongoDB for metadata, PostgreSQL for logs

**2.4 Scalability Features**

- **\*\*Horizontal Scaling\*\***: All services designed to scale horizontally

- **\*\*Auto-Scaling\*\***: HPA configured in Kubernetes manifests

- **\*\*Load Distribution\*\***: RabbitMQ distributes work across multiple workers

- **\*\*Stateless Services\*\***: Upload and Storage services are stateless

**2.5 Data Flow**

1. **\*\*Upload Flow\*\***: User → Upload Service → MongoDB (metadata) → PostgreSQL (logs) → RabbitMQ (job queue)

2. **\*\*Processing Flow\*\***: RabbitMQ → Converter Worker → FFmpeg → Audio file → MongoDB (update) → PostgreSQL (log)

3. **\*\*Download Flow\*\***: User → Storage Service → MongoDB (verify) → Audio file download

---

**3. Remaining Tasks, Challenges, and Risks**

**3.1 Remaining Tasks**

**# High Priority**

1. **\*\*CI/CD Pipeline Setup\*\***

- GitHub Actions workflow for automated builds

- Automated testing in pipeline

- Deployment automation to Kubernetes

- Status: Not started

2. **\*\*Monitoring and Observability\*\***

- Prometheus metrics collection

- Grafana dashboards for visualization

- Application logging aggregation

- Status: Not started

3. **\*\*AWS EKS Deployment\*\***

- Create EKS cluster

- Configure kubectl for EKS

- Deploy all services to production cluster

- Configure LoadBalancers and Ingress

- Status: Not started

**\*\*Medium Priority\*\***

4. **\*\*Web UI Frontend\*\***

- React/HTML interface for file upload

- Progress indicators

- Download interface

- Status: Not started

5. **\*\*Notification Service\*\***

- Email/webhook notifications on completion

- User preference management

- Status: Not started

6. **\*\*Enhanced Testing\*\***

- Integration test suite

- Load/performance testing

- End-to-end test automation

- Status: Partial (unit tests exist)

7. **\*\*Security Enhancements\*\***

- Authentication and authorization

- API rate limiting

- File size/type validation enhancements

- Status: Basic validation implemented

**3.2 Challenges Identified**

**\*\*Technical Challenges\*\***

1. **\*\*Docker Image Build Context\*\***

- Challenge: Converter service needs access to app/ directory

- Solution Implemented: Adjusted build context and Dockerfile paths

- Status: Resolved

2. **\*\*Database Connection Handling\*\***

- Challenge: Handling connection failures and retries

- Current Status: Basic connection handling implemented

- Needs: Connection pooling and retry logic

3. **\*\*File Storage in Kubernetes\*\***

- Challenge: Shared storage across multiple pods

- Solution: Using PersistentVolumeClaims with ReadWriteMany access mode

- Status: Configured

4. **\*\*Service Discovery\*\***

- Challenge: Services need to communicate in Kubernetes

- Solution: Using Kubernetes service DNS names

- Status: Implemented in manifests

**\*\*Operational Challenges\*\***

1. **\*\*FFmpeg Resource Requirements\*\***

- Challenge: Video conversion is CPU and memory intensive

- Solution: Appropriate resource limits in Kubernetes

- Status: Configured, needs tuning

2. **\*\*Error Recovery\*\***

- Challenge: Handling failed conversions and requeuing

- Current Status: Basic error handling, needs improvement

- Needs: Dead letter queue for failed jobs

3. **\*\*Monitoring Integration\*\***

- Challenge: Integrating Prometheus with all services

- Status: Not started, requires instrumentation

**### 3.3 Risks and Mitigation**

**#### \*\*Risk 1: AWS EKS Cost\*\***

- **\*\*Risk\*\***: EKS cluster and associated resources may incur significant costs

- **\*\*Mitigation\*\***:

- Use EKS Fargate for serverless containers

- Implement auto-scaling to scale down during low usage

- Monitor resource usage closely

**#### \*\*Risk 2: Video File Size Limitations\*\***

- **\*\*Risk\*\***: Large video files may cause timeouts or memory issues

- **\*\*Mitigation\*\***:

- Implement file size limits in upload service

- Add streaming for large file processing

- Consider chunked uploads

**#### \*\*Risk 3: Database Performance at Scale\*\***

- **\*\*Risk\*\***: MongoDB and PostgreSQL may become bottlenecks

- **\*\*Mitigation\*\***:

- Implement database connection pooling

- Add caching layer (Redis) if needed

- Monitor query performance

**#### \*\*Risk 4: Message Queue Overflow\*\***

- **\*\*Risk\*\***: RabbitMQ queue may grow too large under heavy load

- **\*\*Mitigation\*\***:

- Scale converter workers based on queue depth

- Implement queue length monitoring

- Set up alerts for queue depth thresholds

**#### \*\*Risk 5: Integration Complexity\*\***

- **\*\*Risk\*\***: Integrating all components may reveal compatibility issues

- **\*\*Mitigation\*\***:

- Test thoroughly in Docker Compose first

- Use feature flags for gradual rollout

- Maintain comprehensive documentation

**### 3.4 Current Status and Next Steps**

- Dockerization- complete

- RabbitMQ, MongoDB, PostgreSQL - configured

- Services integrated with databases- completed

- Application services developed and containerized - completed

- Integration testing in local environment - completed

- Kubernetes deployment ready but not yet deployed to cloud

- CI/CD pipeline - remaining

- Monitoring setup - remaining

- Cloud deployment -remaining

- Final integration and performance testing - remaining

---

**4. Next Steps and Feedback Request**

**4.1 Immediate Next Steps**

1. **\*\*Deploy to AWS EKS\*\***

- Set up EKS cluster

- Push Docker images to ECR

- Deploy using kubectl

2. **\*\*Set Up CI/CD\*\***

- Configure GitHub Actions

- Automate Docker image builds

- Automate Kubernetes deployments

3. **\*\*Implement Monitoring\*\***

- Install Prometheus operator

- Create Grafana dashboards

- Set up alerts

4. **\*\*Integration Testing\*\***

- End-to-end test suite

- Load testing

- Performance optimization

**4.2 Questions for Feedback**

1. **\*\*Architecture\*\***: Is the microservices architecture appropriate for the scale?

2. **\*\*Database Choice\*\***: Is using both MongoDB and PostgreSQL justified, or should we consolidate?

3. **\*\*Deployment Strategy\*\***: Should we use Helm charts for easier deployment management?

4. **\*\*Monitoring Priority\*\***: Which metrics are most critical for demonstrating system health?

5. **\*\*Testing Scope\*\***: What level of integration testing is expected for the final submission?

**### 4.3 Deliverables Status**

- Working prototype (functional locally) - complete

- Architecture documentation - complete

- Docker setup - complete

- Kubernetes manifests - complete

- Cloud deployment (ready but not deployed)

- CI/CD pipeline - remaining

- Monitoring setup - remaining

- Final documentation - remaining

---

**5. Conclusion**

**5.1 Achievements**

We have successfully created a **working prototype** of a microservices-based video-to-audio conversion system that:

- Implements all core functionality

- Demonstrates proper microservices architecture

- Integrates message queues and databases

- Is fully containerized and ready for Kubernetes deployment

- Provides comprehensive API endpoints

**5.2 Readiness**

The system is **ready for deployment** to AWS EKS. All Kubernetes manifests are prepared, services are containerized, and the architecture is sound. The remaining work focuses on:

1. Cloud infrastructure setup (EKS cluster)

2. CI/CD automation

3. Monitoring and observability

4. Production hardening

**5.3 Confidence Level**

**High confidence** in completing the remaining tasks. The foundation is solid, and the remaining tasks are well-defined and achievable.

---

**Prepared by** : Group30