CDN Music App

Project 2
9/29/2025

Name	**Email Address**
Aaron Downing	aaron.downing652@topper.wku.edu
Ryerson Brower	ryerson.brower178@topper.wku.edu

CS 396
Fall 2025
Project Technical Documentation

Contents

List of Figures

1 Introduction

1.1 Project Overview

The Content Delivery Network (CDN) Implementation project is designed to enhance the performance, scalability, and reliability of a music streaming and download service by caching audio content at multiple geographically distributed edge locations. By leveraging edge caching, the system reduces latency, improves download speeds, and provides a seamless user experience for streaming music, even during peak usage periods. The primary objective is to ensure that users can access audio files quickly and efficiently, regardless of their location or device, while the backend intelligently manages routing, caching, and content delivery.

The system architecture was developed using Node.js for the backend, which provides efficient server-side processing, request handling, and real-time updates to cached content. The frontend was built using HTML, CSS, and JavaScript, creating a responsive and intuitive user interface where users can easily find, stream, and download music. Together, this technology stack ensures a fast, scalable, and user-friendly experience.

Functionally, the CDN caches music files in multiple formats (MP3, AAC, OGG) at edge servers to maximize compatibility with diverse devices and playback applications. Requests are routed intelligently based on a user's geographical location and network conditions, ensuring minimal streaming latency. Real-time updates allow new releases and content changes to propagate quickly, guaranteeing that users always have access to the latest music. The system also incorporates analytics to track streaming quality, download speeds, and user interactions, providing administrators with actionable insights.

Non-functional requirements focus on ensuring response times under 100 milliseconds, scalability through the addition of edge servers, and industry-standard security protocols such as encryption and authentication. The system also emphasizes seamless usability with clear feedback and error handling. With built-in monitoring and analytics, the CDN can be continuously optimized for performance and efficiency, making it a robust solution for large-scale music distribution.

1.2 Project Scope

Text goes here.

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
1. The CDN must cache audio files at edge locations to minimize latency for users accessing the music streaming or download service. Cached content should include various formats and qualities to accommodate different user preferences.
2. The system must intelligently route requests to the nearest edge location based on user geographical location and network conditions to ensure optimal performance during music streaming or downloads.
3. The CDN must support real-time content updates, allowing new music releases or changes to existing files to be propagated quickly across edge locations, ensuring that users always access the latest content.
4. The CDN must be capable of serving audio files in multiple formats (e.g., MP3, AAC, OGG) to accommodate various devices and playback requirements, ensuring compatibility with a wide range of client applications.
5. The system must provide analytics features that track user interactions, streaming quality, and download speeds, allowing administrators to monitor performance and make data-driven improvements.
Extended Functional Requirements
1. The CDN must provide users with the ability to adjust playback options, such as play, pause, and skip, during streaming.
2. The system must provide users with the option to download songs for offline playback in supported formats.
3. The CDN must allow users to view recently played or downloaded songs for quick access.

The functional requirements of the CDN music application define the core capabilities the system should provide to deliver music efficiently and effectively to users. The CDN is designed to cache audio files at multiple edge servers, reducing latency and improving performance for both streaming and downloads. Requests are routed to the nearest edge location, ensuring that users receive the fastest possible access based on their geographic location and network conditions. The system also supports real-time updates, so new released and modified files are released across the network. Guaranteeing users always have access to the most up-to-date content. Additionally, the CDN serves audio in multiple formats such as MP3, AAC, and OGG to support a wide range of devices and applications. Users can also download songs for offline playback, access recently played or downloaded content, and control playback with basic options such as play, pause, and skip during streaming.

1.3.2 Non-Functional Requirements

Mandatory Non-Functional Requirements
1. The CDN must ensure that the average response time for streaming and download requests is under 100 milliseconds, even during peak usage periods, to provide a seamless user experience.
2. The system must be designed to handle an increasing number of simultaneous users and requests without performance degradation, allowing for the addition of new edge servers as user demand grows.
3. The system must implement industry-standard security protocols to protect user data and audio content, including encryption of data in transit and robust authentication mechanisms to prevent unauthorized access.
4. The CDN must provide a seamless and intuitive user experience, ensuring that users can easily find, stream, and download music with minimal steps and without confusion, supported by clear feedback and error messaging throughout the process.
5. The system must provide real-time monitoring and analytics capabilities, allowing administrators to track performance metrics, user interactions, and content delivery efficiency, facilitating ongoing optimization and troubleshooting.
Extended Non-Functional Requirements
1. The system must maintain an uptime of at least 99 percent to ensure high availability of the music streaming and download service.
2. The system must maintain error rates (e.g., failed requests) below 1 percent to ensure a consistent and reliable user experience.
3. The CDN must be designed with a responsive user interface so that the service works smoothly on desktops

The non-functional requirements of the CDN project establish the quality standards and operational goals that ensure the system's efficiency, scalability, and reliability. The CDN must provide a fast and seamless experience, maintaining response times under 100 milliseconds, even during peak traffic. Availability is a priority, with the system expected to maintain 99.9 percent uptime, while reliability is ensured by keeping request error rates below 1 percent. Scalability is supported by a design that allows the easy addition of new edge servers as user demand grows. The system also emphasizes security, implementing encryption for data in transit and robust authentication to protect both users and audio content. From a usability perspective, the user interface is designed to be responsive across desktops, tablets, and mobile devices, ensuring accessibility for all. Finally, administrators benefit from real-time monitoring and analytics to track performance, optimize delivery, and troubleshoot issues efficiently.

2 DevOps - Continuous Integration and Continuous Delivery Approach and Results

The approach we took for our CI and CD we setting up GIT Actions. This really helped us be able to test our files the moment they are updated and see if there is any errors. We leverage GitHub Actions to implement our CI workflow, which automatically triggers a series of vital steps upon every code push. This includes running a fast build process and executing automated unit tests and integration tests. Automating these checks allows us to catch errors and conflicts early, significantly reducing the cost and complexity of fixing issues later in the development cycle. The immediate feedback loop established by Git Actions ensures code quality remains high, making the entire release process inherently safer. Our CD practice focuses on creating a reliable, repeatable path to production. The same automated pipeline used for testing will also handle the process of packaging the application and preparing it for release to staging or production environments. This standardization removes human error and guarantees consistency across deployments, making releases safer and more predictable. This highly reliable, automated process is the primary driver for achieving a faster time to market for new music content and CDN features.

3 DevOps - Architecture Approach, Models, and Results

Our CDN project uses a loosely coupled design, where each part of the system is separated into clear components. This modular setup makes it easier to work on parts independently, release new features often, and scale the system efficiently while keeping it reliable. It also helps us deliver value faster.

Content Origin: The main source of audio files (the music folder). Its job is only to store and serve content, separate from delivery logic.

Delivery and Routing Layer: Simulated by the Edge Cache and Geo-Routing in server.js, this layer manages caching, routing, and time-to-live (TTL) updates.

Analytics and Monitoring Service: This tracks performance data like latency and cache hits through the /analytic endpoint, keeping monitoring separate from content delivery.

Because these parts are independent, we can update caching or routing without touching the content library. This makes deployments safer and faster, while keeping system efficiency high.

4 DevOps - Product and Process Approach and Results

Our approach to building the Content Delivery Network (CDN) for the music streaming service focuses on being customer-centered and guided by feedback. At each stage, we put the user experience first, making sure streaming has low delay, downloads are fast, and songs can be played smoothly in different formats.

We built feedback into the development process using performance tracking, user activity data, and direct feedback, which helped us improve caching, routing, and delivery rules step by step. To support steady progress and reliable updates, we used GitHub Actions for automated testing, integration, and deployment. This allowed quick changes while keeping quality and stability. By combining user feedback, teamwork, and DevOps tools, our CDN keeps improving to give better user experiences and more efficient content delivery.

5 DevOps - Product Management and Monitoring Approach and Results

For our music streaming CDN, we set up monitoring and feedback systems that give real-time information about system performance. We track metrics like streaming latency, cache hits, download speeds, and errors to see how the system is working at all times.

Dashboards and alerts help the team spot problems early, so we can fix them before users are affected. We also log user interactions and system behavior to guide improvements in caching, routing, and content delivery.

This approach helps keep the system stable, makes decision-making easier, and allows the team to continuously improve performance. By using real-time data and feedback, we ensure a better experience for users and more efficient operations for the team.

6 DevOps - Cultural Approach and Results

Our team focused on creating a collaborative environment built on trust and shared responsibility. Everyone had a voice in decisions, and we encouraged open communication to make sure ideas and concerns were heard.

We emphasized continuous learning by sharing knowledge, reviewing code together, and experimenting with new approaches to improve our CDN and music streaming service. Mistakes were treated as learning opportunities, not failures, which allowed the team to try innovative solutions without fear.

This culture of trust, collaboration, and experimentation helped the team perform at a high level. By supporting each other, sharing responsibility, and continuously learning, we were able to develop better solutions, improve system performance, and deliver a stronger user experience.

7 Software Testing and Results

7.1 Unit Testing Plan

Test Plan Identifier: UT-001

Introduction: Unit testing ensures that individual modules of the CDN (e.g., caching logic, routing function, API endpoints) work correctly in isolation. The objective is to validate correctness, identify defects early, and confirm that each component meets its design specifications.

Test item: The Software Under Test includes caching functions, request-routing algorithms, content update methods, and analytics data collectors. Each module is tested independently before integration with other components.

Features to test/not to test: In-scope features are caching rules, request handlers, and analytics logging. Out-of-scope are UI-related functions, as they depend on system-level behavior rather than individual modules.

Approach: Unit tests use automated frameworks (JUnit, Jest) to validate logic. Tests check cache hits/misses, request routing accuracy, and error handling. Mock data simulates user requests and system responses.

Test deliverables: Deliverables include automated test scripts, execution logs, and coverage reports that highlight tested code areas and defect detection rates.

Item pass/fail criteria: A unit passes if outputs match expected values under valid and invalid inputs. Failures occur when logic does not meet design requirements.

Environmental needs: Local developer environments with IDEs, version control integration (GitHub), and automated CI/CD pipelines with GitHub Actions.

Responsibilities: The team is responsible for writing and maintaining unit tests and reviewing coverage.

Staffing and training needs: The team requires familiarity with automated testing tools, mocking frameworks, and writing high-quality, maintainable test cases.

Schedule: Unit tests are written alongside development and must pass before merging to the main branch. Execution occurs daily in CI/CD pipelines.

Risks and Mitigation: Risks include incomplete coverage and overlooked edge cases. Mitigation includes peer code reviews, automated coverage analysis, and enforcing minimum thresholds.

Approvals: Team approval required.

7.2 Integration Testing Plan

Test Plan Identifier: IT-001

Introduction: Integration testing verifies that modules (e.g., caching, routing, and content updates) interact correctly when combined. The goal is to uncover interface issues and confirm data flow between components.

Test item: CDN caching layer, request-routing engine, analytics collector, and content update service working together in staging environments.

Features to test/not to test: In-scope: cache-to-routing communication, routing-to-analytics logging, and content updates across modules. Out-of-scope: full-scale performance, which belongs to system testing.

Approach: Integration testing combines automated API tests with manual scenario testing to confirm modules exchange data correctly and maintain expected behavior.

Test deliverables: Integration test cases, execution results, error reports, and a final integration test summary.

Item pass/fail criteria: Pass if data is transferred and processed without corruption. Fail if communication breaks or modules produce inconsistent results.

Environmental needs: Staging environment with partial deployment, test CDN nodes, and monitoring tools for log verification.

Responsibilities: The team leads integration execution and debugging identified defects.

Staffing and training needs: Team members must be comfortable with API testing, staging deployment setup, and debugging distributed systems.

Schedule: Conducted after unit testing, with weekly runs during development sprints and before feature merges.

Risks and Mitigation: Risk of mismatched APIs or unexpected data formats. Mitigation through interface contracts, schema validation, and continuous integration checks.

Approvals: Team approval required.

7.3 System Testing Plan

Test Plan Identifier: ST-001

Introduction: System testing validates the complete CDN as a whole, ensuring that functional and non-functional requirements are met in realistic conditions.

Test item: Full CDN deployment including caching, routing, content update, analytics, and user-facing streaming/download features.

Features to test/not to test: In-scope: latency under load, multi-format streaming (MP3, AAC, OGG), and security protocols. Out-of-scope: unrelated UI features not tied to CDN delivery.

Approach: System tests combine automated load testing, performance measurement, and manual validation of streaming/download processes. Both functional and non-functional requirements are tested.

Test deliverables: Deliverables include test cases, performance reports, latency graphs, error logs, and system validation summaries.

Item pass/fail criteria: Pass if requirements (e.g., ¡100 ms response, secure content delivery) are met under expected loads. Fail if KPIs fall below defined thresholds.

Environmental needs: Full deployment environment with edge servers, simulated global traffic, and monitoring dashboards.

Responsibilities: The team executes tests, sets up infrastructure, and fixes identified issues.

Staffing and training needs: The team requires training in load testing tools (JMeter, Locust) and monitoring tools (Grafana, Prometheus).

Schedule: Executed after integration tests and before user acceptance testing. Scheduled for pre-release builds.

Risks and Mitigation: Risks include high load causing server crashes or security vulnerabilities being missed. Mitigation includes stress testing beyond peak loads and automated vulnerability scans.

Approvals: Team approval required.

7.4 Acceptance Testing Plan

Test Plan Identifier: AT-001

Introduction: Acceptance testing ensures the CDN meets end-user needs and business goals. It validates that the service performs as expected for real-world use cases.

Test item: Full CDN service in near-production conditions, tested from an end-user perspective for streaming, downloading, and reliability.

Features to test/not to test: In-scope: streaming performance, download reliability, ease of access, and error messaging. Out-of-scope: developer-only internal APIs not exposed to users.

Approach: User-focused testing conducted with manual scenario execution, feedback collection, and trial runs across devices and locations.

Test deliverables: Deliverables include user acceptance criteria, test case execution results, feedback reports, and a final acceptance sign-off.

Item pass/fail criteria: Pass if end-user requirements and business goals are met. Fail if critical functionality or usability concerns remain unresolved.

Environmental needs: Production-like environment with distributed CDN nodes

8 Conclusion

In conclusion this project went a bit smoother than the first project we did. This may be because we jumped on it a lot faster than the other. We were able to get the hardest part which was the coding done pretty fast. One thing that we could have improved on was the way we implemented the Music App. We not to sure how to make a complete network for the music app so we only used a local host. Similarly for the edge systems we only simulated it with our code that gave us the response time and the predicted edge system, such as New York, Asia, and more. We downloaded two songs, a longer song (around 7 mins) and a shorter song (around 3 mins). Some thing we didn't know how to implement was the ability to download and upload song remotely. So there was a lot we wished we could have polished out but we had to keep our scope pretty small to complete this project. But overall thanks to this project brought a lot of light to what DevOps was and how to use it to our advantage. We will be able to use these skills in the future.

9 Appendix

9.1 Software Product Build Instructions

Download code along with node js. In powershell run npm install and npn start to get the local host. Then navigate to the localhost.

9.2 Software Product User Guide

Download code along with node js. In powershell run npm install and npn start to get the local host. Then navigate to the localhost.

9.3 Source Code with Comments

server.js

```
const express = require('express');
const path = require('path');
const fs = require('fs');

// Using a simple object for in-memory cache simulation
const edgeCache = new Map();
// Cache TTL (Time To Live) in milliseconds (e.g., 5 minutes)
const CACHE_TTL = 5 * 60 * 1000;

// --- CDN Simulation: Edge Locations ---
// In a real CDN, these would be separate servers. Here, we simulate by defining
// the locations and assigning a request to one of them.
const EDGE_LOCATIONS = [
    { id: 'NYC-01', region: 'North America', latencyMs: 50 },
    { id: 'LON-05', region: 'Europe', latencyMs: 80 },
    { id: 'TOK-12', region: 'Asia', latencyMs: 150 }
];


function simulateGeoRouting(request) {
    const randomIndex = Math.floor(Math.random() * EDGE_LOCATIONS.length);
    const selectedEdge = EDGE_LOCATIONS[randomIndex];

    console.log(`[Geo-Routing] Request routed to Edge Location: ${selectedEdge.id} (${
        selectedEdge.region})`);

    return selectedEdge;
}
```

```
const app = express();
const port = 3000;

// Set up a path for the music files
const musicDir = path.join(__dirname, 'music');

app.use(express.static(path.join(__dirname, 'public')));

// Middleware to start timing the request (for response time measurement)
app.use((req, res, next) => {
    req.requestStartTime = Date.now();
    next();
});

// Endpoint to get the list of songs available
app.get('/songs', (req, res) => {
    fs.readdir(musicDir, (err, files) => {
        if (err) {
            console.error('Failed to read music directory:', err);
            return res.status(500).send('Error reading music files.');
        }

        const audioFiles = files.filter(file => {
            const ext = path.extname(file).toLowerCase();
            return ['.mp3', '.aac', '.ogg'].includes(ext);
        });

        res.json(audioFiles);
    });
});

// Endpoint to serve a specific song file with caching and routing logic
app.get('/music/:songName', (req, res) => {
    const songName = req.params.songName;
    const songPath = path.join(musicDir, songName);

    // 1. Simulate Intelligent Routing (Functional Req 2)
    const edgeLocation = simulateGeoRouting(req);

    // 2. Check for Directory Traversal
    if (!songPath.startsWith(musicDir)) {
        console.error(`Security Warning: Directory traversal attempt for ${songName}`)
            ;
        return res.status(400).send('Invalid file path.');
    }

    // 3. Caching Logic (Functional Req 1)
    const cacheKey = `${edgeLocation.id}:${songName}`;
    const cachedItem = edgeCache.get(cacheKey);

    if (cachedItem && Date.now() < cachedItem.expiry) {
        // Cache Hit
        console.log(`[Edge Cache] Hit: Serving ${songName} from in-memory cache on ${
            edgeLocation.id}.`);

        // Non-Functional Req 1: Add simulated latency for performance measurement
        setTimeout(() => {
            res.setHeader('Content-Type', cachedItem.mimeType);
            res.send(cachedItem.data);
            // Measure and Log Response Time
            const responseTime = Date.now() - req.requestStartTime;
            console.log(`[Performance Metric] Response Time for ${songName}: ${
               responseTime}ms (Target: <100ms)`);
        }, edgeLocation.latencyMs / 2); // Divide by 2 to keep the first hop fast

        return;
    }

    // Cache Miss: Read from origin (disk)
    fs.readFile(songPath, (err, data) => {
        if (err) {
            console.error(`Failed to read file ${songName} from disk:`, err);
            return res.status(404).send('Song not found.');
        }

        // Determine MIME Type for proper streaming
        const mimeType = express.static.mime.lookup(songName) || 'audio/mpeg';

        // Store in Cache (Simulating Edge Storage)
        edgeCache.set(cacheKey, {
            data: data,
            mimeType: mimeType,
            expiry: Date.now() + CACHE_TTL,
            // Functional Req 3: Real-time update check (by expiration)
            createdAt: new Date().toISOString()
        });

        console.log(`[Edge Cache] Miss: Caching file ${songName} and serving from
           origin on ${edgeLocation.id}.`);

        // Serve the newly loaded file
        // Non-Functional Req 1: Add simulated latency for performance measurement
        setTimeout(() => {
            res.setHeader('Content-Type', mimeType);
            res.send(data);

            // Measure and Log Response Time
            const responseTime = Date.now() - req.requestStartTime;
            console.log(`[Performance Metric] Response Time for ${songName}: ${
               responseTime}ms (Target: <100ms)`);
        }, edgeLocation.latencyMs);

    });
});

// Serve the static HTML file
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});

// Start the server
app.listen(port, () => {
    console.log(`Server is running at http://localhost:${port}`);
    console.log('--- Edge CDN Simulation Initialized ---');
    console.log(`Cache TTL set to ${CACHE_TTL / 1000} seconds.`);
    console.log('----------------------------------------');
});
```

script.js

```
const songListElement = document.getElementById('song-list');
const audioPlayer = document.getElementById('audio-player');
const songTitleElement = document.getElementById('song-title');

const fetchSongs = async () => {
    try {
        const response = await fetch('/songs');
        if (!response.ok) {
            throw new Error('Failed to fetch songs');
        }
        const songs = await response.json();
        renderSongs(songs);
    } catch (error) {
        console.error('Error:', error);
        songListElement.innerHTML = '<li class="text-center text-red-400">Failed to
           load songs. Make sure the server is running.</li>';
    }
};

const renderSongs = (songs) => {
    songListElement.innerHTML = '';
    if (songs.length === 0) {
        songListElement.innerHTML = '<li class="text-center text-gray-400">No songs
           found. Add files to the `music` directory.</li>';
        return;
    }

    songs.forEach(song => {
        const listItem = document.createElement('li');
        listItem.className = 'music-list-item p-3 rounded-lg flex items-center justify
           -between transition-all duration-200 bg-gray-800 border border-transparent
            hover:border-indigo-500';
        listItem.innerHTML = `
            <span class="truncate">${song}</span>
            <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" fill="
               currentColor" class="w-5 h-5 text-indigo-400 ml-2 flex-shrink-0">
                <path fill-rule="evenodd" d="M4.5 5.653c0-1.23.955-2.23 2.13-2.23c.319
                    0 .633.072.923.211l10.966 5.378a2.25 2.25 0 0 1 0 4.026l-10.966
                   5.378c-.29.138-.604.21-.923.21c-1.175 0-2.13-.999-2.13-2.23V5.653Z
                   " clip-rule="evenodd" />
            </svg>
        `;
        listItem.addEventListener('click', () => {
            const songUrl = `/music/${encodeURIComponent(song)}`;
            audioPlayer.src = songUrl;
            songTitleElement.textContent = song;
        });
        songListElement.appendChild(listItem);
    });
};

// Fetch and display songs when the page loads
document.addEventListener('DOMContentLoaded', fetchSongs);
```

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Music CDN App</title>
    <link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;600;700&
       display=swap" rel="stylesheet">
    <script src="https://cdn.tailwindcss.com"></script>
    <link rel="stylesheet" href="styles.css">
</head>
<body class="bg-gray-900 text-gray-100 min-h-screen flex items-center justify-center">
    <div class="container bg-gray-800 rounded-2xl shadow-2xl p-8 m-4 w-full">
        <h1 class="text-4xl font-bold text-center mb-6 text-indigo-400">Music CDN
           Player</h1>

        <!-- Media Player -->
        <div class="player-container mb-6 p-4 bg-gray-700 rounded-xl shadow-lg">
            <h2 id="song-title" class="text-xl font-semibold text-center text-gray-200
                mb-2 truncate">Select a song</h2>
            <audio id="audio-player" class="w-full" controls autoplay></audio>
        </div>

        <!-- Song List -->
        <div class="song-list-container bg-gray-700 rounded-xl shadow-inner p-4">
            <h3 class="text-lg font-bold mb-3 text-center text-gray-200">Available
               Songs</h3>
            <ul id="song-list" class="space-y-2">
                <!-- Song items will be dynamically added here -->
            </ul>
        </div>
    </div>

    <script src="script.js"></script>
</body>
</html>
```