CineFlux-AutoXML Build Optimization

This document outlines the build optimization strategies implemented for the CineFlux-AutoXML application to meet the specified performance targets.

Performance Targets

- Initial load time: Under 2.5 seconds on standard broadband
- Time to interactivity: Under 3.5 seconds
- Main bundle size: Under 300KB (gzipped)
- Efficient WebAssembly loading

Optimization Strategies Implemented

1. Vite Configuration Optimizations

The `vite.config.ts` file has been updated with the following optimizations:

- **Environment-specific builds**: Separate configurations for development and production environments
- **Source map control**: Disabled source maps in production to reduce bundle size
- **Minification**: Enabled terser minification with console removal in production
- **Code splitting**: Implemented strategic code splitting for:
  - WebAssembly-related modules (`wasm-modules` chunk)
  - React core libraries (`react-vendor` chunk)
  - UI components (`ui-vendor` chunk)
  - Audio processing libraries (`audio-vendor` chunk)
  - Other dependencies (`vendor` chunk)
- **Asset optimization**: Configured proper asset handling with hashed filenames for better caching
- **Modern JavaScript**: Targeting modern browsers with `esnext` to reduce transpilation overhead

2. Environment Variable Configuration

Created environment-specific configuration files:

- `.env.development`: Development-specific variables
- `.env.production`: Production-specific variables
- `.env.example`: Template for required variables

Implemented the following environment variables: - `NODE_ENV`: Controls development/production mode - `VITE_APP_VERSION`: Application version for tracking - `VITE_WASM_CDN_URL`: Configurable CDN for WebAssembly modules - `VITE_FEATURE_FLAGS`: JSON string for enabling/disabling experimental features

3. **Code Splitting Implementation**

   - **Dynamic imports**: Used React.lazy for route-based code splitting
   - **Component chunking**: Implemented logical component chunking with custom chunk names
   - **Lazy loading**: Added suspense boundaries for non-critical components
   - **Preloading**: Implemented preloading for critical WebAssembly modules

4. **WebAssembly Optimization**

   - **Efficient loading**: Created a dedicated WebAssembly loader utility (`wasmLoader.ts`)
   - **Caching strategy**: Implemented module caching to prevent duplicate loading
   - **Progress tracking**: Added support for loading progress indicators
   - **CDN integration**: Configured CDN support for WebAssembly modules in production
   - **Preloading**: Added preloading for critical WebAssembly modules

5. **Feature Flag System**

   - **Environment-based configuration**: Feature flags controlled via environment variables
   - **Type-safe access**: Created a utility for type-safe feature flag access
   - **Default fallbacks**: Implemented sensible defaults for missing flags

Build Scripts

The following npm scripts have been added for building and analyzing:

- `npm run build:dev`: Build for development environment
- `npm run build:prod`: Build for production environment
- `npm run build:analyze`: Build with bundle analysis
- `npm run analyze`: Analyze bundle size and composition
- `npm run clean`: Clean the build directory

Bundle Analysis

The bundle analysis script (`scripts/analyze-bundle.js`) provides:

- Detailed breakdown of chunk sizes
- Gzipped size calculations
- Performance assessment against targets
- Recommendations for further optimization

Usage Instructions

1. **Development Build**:

```
npm run build:dev
```

2. **Production Build**:

```
npm run build:prod
```

3. **Analyze Bundle**:

```
npm run build:analyze
npm run analyze
```

Performance Monitoring

To ensure the application meets the performance targets:

1. Run the bundle analysis after each significant change
2. Monitor the main bundle size to keep it under 300KB (gzipped)
3. Test initial load time and time to interactivity on various network conditions
4. Optimize WebAssembly loading for specific media processing operations

Future Optimization Opportunities

- Further component splitting for large components
- Implement HTTP/2 server push for critical resources
- Add service worker for offline support and caching
- Implement resource hints (preload, prefetch) for critical assets
- Consider using image CDN for media assets