Research Document: Desktop Video Converter

Date: September 16, 2025
Feature: Desktop Video Converter - React + Electron + Vite Integration
Status: Phase 0 Complete

Executive Summary

This research document consolidates findings for implementing a Desktop Video Converter using React + Electron + Vite stack with shadcn/ui for professional UI components. Key decisions focus on security, performance, maintainability, and user experience for a Windows desktop application that converts videos to web-optimized MP4 format.

1. React + Electron + Vite Integration

Decision: electron-vite as Primary Build Tool

Rationale: electron-vite provides pre-configured Electron support with Vite's fast HMR, optimized asset handling, and TypeScript decorators support.

Key Benefits:

- ⚡ Vite-powered with fast HMR for renderer processes
- 🔥 Hot reloading for main process and preload scripts
- 🛠️ Pre-configured for Electron (no complex setup needed)
- 📦 Out-of-the-box TypeScript, React support
- 🔒 Built-in source code protection with bytecode compilation

Implementation Pattern:

```
// electron.vite.config.ts
import { defineConfig, externalizeDepsPlugin } from 'electron-vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  main: {
    plugins: [externalizeDepsPlugin()]
  },
  preload: {
    plugins: [externalizeDepsPlugin()]
  },
  renderer: {
    plugins: [react()],
    resolve: {
      alias: {
        '@': path.resolve(__dirname, './src')
      }
    }
```


Dr. Rasika Kulasinghe

```
  }
})
```

Security Pattern: Context isolation + preload scripts

```
// preload/index.ts
import { contextBridge, ipcRenderer } from 'electron'

contextBridge.exposeInMainWorld('electronAPI', {
  selectFile: () => ipcRenderer.invoke('select-file'),
  convertVideo: (options) => ipcRenderer.invoke('convert-video', options),
  onProgress: (callback) => ipcRenderer.on('conversion-progress', callback)
})
```

Alternatives Considered:

- Electron Forge: More complex setup, less Vite optimization
- Custom Vite + Electron setup: Requires manual configuration
- electron-builder only: Missing development server integration

2. FFmpeg Integration for Electron

Decision: fluent-ffmpeg + Static Binary Distribution

Rationale: fluent-ffmpeg provides robust Node.js FFmpeg integration with comprehensive progress tracking, while static binaries ensure consistent deployment.

Key Benefits:

- 📊 Built-in progress tracking via `progress` event
- 🔧 Fluent API for complex video operations
- 📦 Static binary distribution eliminates runtime dependencies
- ⚡ Real-time conversion monitoring
- 🛡️ Error handling and process control (kill, renice)

Implementation Pattern:

```
// services/video-converter.ts
import ffmpeg from 'fluent-ffmpeg'
import ffmpegPath from 'ffmpeg-static'

ffmpeg.setFfmpegPath(ffmpegPath)

export class VideoConverter {
  convert(inputPath: string, outputPath: string, onProgress: (progress) =>
void) {
    return new Promise((resolve, reject) => {
```

```
        ffmpeg(inputPath)
          .videoCodec('libx264')
          .audioCodec('aac')
          .format('mp4')
          .videoFilters(['scale=trunc(iw/2)*2:trunc(ih/2)*2'])
          .on('progress', onProgress)
          .on('end', resolve)
          .on('error', reject)
          .save(outputPath)
      })
    }
  }
```

Progress Tracking Pattern:

```
ffmpeg(inputPath)
  .on('progress', (progress) => {
    // progress.percent, progress.frames, progress.currentFps
    mainWindow.webContents.send('conversion-progress', {
      percent: progress.percent,
      frames: progress.frames,
      currentFps: progress.currentFps,
      timemark: progress.timemark
    })
  })
```

Binary Distribution:

- Use `ffmpeg-static` package for bundled binaries
- Package-specific binaries for Windows x64
- ASAR unpacking for executable access

Alternatives Considered:

- FFmpeg.wasm: Browser-based, limited performance for desktop
- Native FFmpeg installation: Deployment complexity
- Other Node.js wrappers: Less mature progress tracking

3. shadcn/ui Setup for Electron

Decision: shadcn/ui with Tailwind CSS in Renderer Process

Rationale: shadcn/ui provides professional, accessible components that work seamlessly in Electron renderer processes with proper Tailwind configuration.

Key Benefits:

- 🎨 Professional, modern component library
- ♿ WCAG AA accessibility compliance built-in

- 🔧 Copy-paste components, full customization
- ⚡ Optimized for React + TypeScript
- 🎯 Perfect for desktop application UI patterns

Setup Pattern:

```
# Initialize shadcn/ui in renderer
cd src/renderer
pnpm dlx shadcn@latest init
pnpm dlx shadcn@latest add button card progress dialog
```

Configuration (components.json):

```
{
  "$schema": "https://ui.shadcn.com/schema.json",
  "style": "new-york",
  "rsc": false,
  "tsx": true,
  "tailwind": {
    "config": "tailwind.config.js",
    "css": "src/renderer/src/index.css",
    "baseColor": "neutral",
    "cssVariables": true
  },
  "aliases": {
    "components": "@/components",
    "utils": "@/lib/utils"
  }
}
```

Electron-Specific Considerations:

- No server-side rendering (rsc: false)
- Path aliases work with Vite resolver
- CSS variables for theming work in Electron context
- All components render in single renderer process

Component Usage Pattern:

```
import { Button } from '@/components/ui/button'
import { Progress } from '@/components/ui/progress'
import { Card, CardContent, CardHeader, CardTitle } from '@/components/ui/card'

export function VideoConverter() {
  return (
    <Card>
      <CardHeader>
```

```
        <CardTitle>Video Conversion</CardTitle>
      </CardHeader>
      <CardContent>
        <Progress value={progressPercent} />
        <Button onClick={handleConvert}>Convert</Button>
      </CardContent>
    </Card>
  )
}
```

Alternatives Considered:

- Material-UI: Heavier bundle, React-focused
- Chakra UI: Good but less Windows-native feel
- Custom components: Higher development time
- Ant Design: Less modern, heavier

4. Video Format Detection and Validation

Decision: File Extension + FFprobe Validation

Rationale: Two-tier validation provides user-friendly file filtering with robust format verification using FFmpeg's probe functionality.

Implementation Strategy:

1. **Primary**: Electron dialog filters for common formats
2. **Secondary**: FFprobe metadata validation for file integrity

File Dialog Pattern:

```
// main/ipc-handlers.ts
ipcMain.handle('select-file', async () => {
  const result = await dialog.showOpenDialog({
    properties: ['openFile'],
    filters: [
      {
        name: 'Video Files',
        extensions: ['mp4', 'avi', 'mov', 'mkv', 'wmv', 'flv', 'webm', 'm4v']
      },
      { name: 'All Files', extensions: ['*'] }
    ]
  })
  return result.filePaths
})
```

Validation Pattern:

```typescript
// services/video-validator.ts
import ffmpeg from 'fluent-ffmpeg'

export async function validateVideoFile(filePath: string): Promise<{
  isValid: boolean
  metadata?: any
  error?: string
}> {
  return new Promise((resolve) => {
    ffmpeg.ffprobe(filePath, (err, metadata) => {
      if (err) {
        resolve({ isValid: false, error: err.message })
      } else {
        const hasVideoStream = metadata.streams.some(
          stream => stream.codec_type === 'video'
        )
        resolve({
          isValid: hasVideoStream,
          metadata: hasVideoStream ? metadata : undefined,
          error: hasVideoStream ? undefined : 'No video stream found'
        })
      }
    })
  })
}
```

Error Handling Pattern:

- File extension mismatch: User-friendly dialog
- Corrupted files: FFprobe error with recovery options
- Unsupported codecs: Clear error messages with format suggestions

Alternatives Considered:

- File-type detection libraries: Limited video-specific features
- Magic number detection: Less reliable than FFprobe
- MediaInfo: Additional dependency complexity

5. Real-time Progress Tracking Patterns

Decision: IPC-based Progress Streaming

Rationale: Leverage Electron's IPC for real-time progress updates from main process (FFmpeg) to renderer (UI), ensuring responsive user interface.

Architecture Pattern:

```
Main Process (FFmpeg) → IPC Messages → Renderer Process (React UI)
```

Implementation Pattern:

```
// Main process - video conversion service
class VideoConversionService {
  async convert(input: string, output: string, mainWindow: BrowserWindow) {
    const command = ffmpeg(input)
      .on('progress', (progress) => {
        mainWindow.webContents.send('conversion-progress', {
          percent: Math.round(progress.percent || 0),
          frames: progress.frames,
          currentFps: progress.currentFps,
          currentKbps: progress.currentKbps,
          targetSize: progress.targetSize,
          timemark: progress.timemark,
          stage: 'converting'
        })
      })
      .on('end', () => {
        mainWindow.webContents.send('conversion-complete', { success: true })
      })
      .on('error', (err) => {
        mainWindow.webContents.send('conversion-error', { error: err.message })
      })

    return command.save(output)
  }
}
```

React Hook Pattern:

```
// Renderer process - progress hook
export function useVideoConversion() {
  const [progress, setProgress] = useState(0)
  const [stage, setStage] = useState<'idle' | 'converting' | 'complete' |
'error'>('idle')

  useEffect(() => {
    const handleProgress = (_, data) => {
      setProgress(data.percent)
      setStage(data.stage)
    }

    window.electronAPI.onProgress(handleProgress)
    return () => window.electronAPI.removeListener('conversion-progress',
handleProgress)
  }, [])

  return { progress, stage }
}
```

UI Component Pattern:

```
import { Progress } from '@/components/ui/progress'

export function ConversionProgress() {
  const { progress, stage } = useVideoConversion()

  return (
    <div className="space-y-2">
      <Progress value={progress} className="w-full" />
      <p className="text-sm text-muted-foreground">
        {stage === 'converting' ? `Converting... ${progress}%` : 'Ready'}
      </p>
    </div>
  )
}
```

Performance Optimizations:

- Throttle progress updates (max 2 updates/second)
- Buffer progress data to prevent UI freezing
- Graceful degradation if progress unavailable

Alternatives Considered:

- Polling-based progress: Less efficient, higher latency
- WebSocket communication: Unnecessary complexity
- File-based progress sharing: I/O overhead

6. Single Executable Packaging

Decision: electron-builder with NSIS Installer

Rationale: electron-builder provides comprehensive packaging with portable executable options and NSIS installer for professional Windows distribution.

Configuration Pattern:

```
// electron-builder config
{
  "appId": "com.videoconverter.desktop",
  "productName": "Desktop Video Converter",
  "directories": {
    "output": "dist-packages"
  },
  "files": [
    "dist-electron/**/*",
    "dist/**/*",
```

```
      "node_modules/**/*",
      "package.json"
    ],
    "win": {
      "target": [
        {
          "target": "nsis",
          "arch": ["x64"]
        },
        {
          "target": "portable",
          "arch": ["x64"]
        }
      ],
      "icon": "build/icon.ico",
      "artifactName": "${productName}-${version}-${arch}.${ext}"
    },
    "nsis": {
      "oneClick": false,
      "allowToChangeInstallationDirectory": true,
      "artifactName": "${productName}-Setup-${version}.${ext}"
    },
    "portable": {
      "artifactName": "${productName}-Portable-${version}.${ext}"
    }
  }
```

Build Script Pattern:

```
{
  "scripts": {
    "build": "electron-vite build",
    "build:win": "npm run build && electron-builder --win",
    "build:portable": "npm run build && electron-builder --win portable",
    "dist": "npm run build && electron-builder --publish=never"
  }
}
```

Asset Bundling:

- ASAR packaging for source protection
- FFmpeg binaries in unpacked resources
- Icon and manifest embedding
- Code signing (future consideration)

Distribution Options:

1. **Portable Executable**: Single .exe file, no installation
2. **NSIS Installer**: Traditional Windows installer

3. **Auto-updater Integration**: Future enhancement capability

Alternatives Considered:

- Electron Forge: Less Windows-focused packaging
- Manual packaging: Complex and error-prone
- Tauri: Different framework, Rust-based

Technical Architecture Summary

Project Structure Decision

```
src/
├── main/                # Electron main process
│   ├── index.ts         # Application entry point
│   ├── window.ts        # Window management
│   ├── ipc/             # IPC handlers
│   └── services/        # Video conversion, file operations
├── renderer/            # React frontend
│   ├── src/
│   │   ├── components/  # shadcn/ui components
│   │   ├── hooks/       # React hooks for Electron integration
│   │   ├── lib/         # Utilities and helpers
│   │   └── pages/       # Application screens
├── preload/             # Security bridge
│   └── index.ts         # Context bridge setup
└── shared/              # Common types and constants
    ├── types/           # TypeScript interfaces
    └── constants/       # Application constants
```

Key Dependencies

- **electron-vite**: Build tool and development server
- **fluent-ffmpeg**: Video processing library
- **ffmpeg-static**: Bundled FFmpeg binaries
- **shadcn/ui**: UI component library
- **@radix-ui/react-***: Component primitives
- **tailwindcss**: Utility-first CSS framework
- **electron-builder**: Application packaging

Development Workflow

1. **Development**: `electron-vite dev` (HMR enabled)
2. **Testing**: `vitest` for unit tests, Playwright for E2E
3. **Building**: `electron-vite build` → `electron-builder`
4. **Distribution**: Portable .exe or NSIS installer

Risk Mitigation

Security Considerations

- Context isolation enabled
- Node integration disabled in renderer
- Sandbox mode for renderer process
- Preload script as security bridge
- Input validation for all file operations

Performance Considerations

- FFmpeg process isolation
- Memory usage monitoring
- Progress update throttling
- Temporary file cleanup
- Large file handling (>2GB)

Maintenance Considerations

- Modular architecture for feature expansion
- Type-safe IPC communication
- Comprehensive error logging
- Automated testing pipeline
- Version management strategy

Conclusion

The selected technology stack (React + Electron + Vite + shadcn/ui + FFmpeg) provides a robust foundation for the Desktop Video Converter application. Key decisions prioritize:

1. **Developer Experience**: Fast development with HMR and TypeScript
2. **User Experience**: Professional UI with accessible components
3. **Performance**: Efficient video processing with real-time feedback
4. **Security**: Electron best practices with context isolation
5. **Maintainability**: Modular architecture with clear separation of concerns
6. **Distribution**: Professional packaging with multiple deployment options

This research provides the technical foundation for Phase 1 implementation planning.