Implementation Plan: Desktop Video Converter

Branch: `001-desktop-video-converter` | **Date**: September 16, 2025 | **Spec**: spec.md
Input: Feature specification from `/specs/001-desktop-video-converter/spec.md`

Execution Flow (/plan command scope)

```
1. Load feature spec from Input path
   → ☑ Loaded successfully
2. Fill Technical Context (scan for NEEDS CLARIFICATION)
   → ☑ React + Electron + Vite stack specified
   → ☑ shadcn UI library for components
3. Evaluate Constitution Check section below
   → ☑ Initial check complete
4. Execute Phase 0 → research.md
   → 🔄 IN PROGRESS
5. Execute Phase 1 → contracts, data-model.md, quickstart.md, agent-specific
template file
6. Re-evaluate Constitution Check section
7. Plan Phase 2 → Describe task generation approach (DO NOT create tasks.md)
8. STOP - Ready for /tasks command
```

Summary

Desktop Video Converter: A Windows desktop application that converts video files to web-optimized MP4 format using Electron framework with React frontend, featuring drag-and-drop file selection, real-time progress tracking, and packaged as a portable executable. Technical approach uses React + Electron + Vite with shadcn UI for professional appearance and FFmpeg for video processing.

Technical Context

Language/Version: TypeScript 5.0+, Node.js 18+
Primary Dependencies: Electron 27+, React 18+, Vite 5+, shadcn/ui, FFmpeg, Tailwind CSS
Storage: Local file system operations, temporary file handling
Testing: Vitest, Playwright for E2E, Jest for Electron main process
Target Platform: Windows 10/11 desktop (x64)
Project Type: Desktop application (Electron)
Performance Goals: Real-time progress updates, efficient video processing, <2GB memory usage
Constraints: Single portable .exe file, no installation required, WCAG AA accessibility
Scale/Scope: Single-user desktop application, support for common video formats (MP4, AVI, MOV, MKV)

User-provided technical details: Use React + Electron + Vite. Use best practice according to their official documentations. Well organized project structure, later expandable while adding extra features. Do research to select most appropriate required libraries. UI design must have well organized, professional looking appearance. Use shadcn UI library to implement UI.

Constitution Check

GATE: Must pass before Phase 0 research. Re-check after Phase 1 design.

Simplicity:

- Projects: 3 (main electron process, renderer react app, shared types)
- Using framework directly? Yes (React, Electron, Vite without wrappers)
- Single data model? Yes (video conversion job state)
- Avoiding patterns? Yes (direct FFmpeg integration, no over-abstraction)

Architecture:

- EVERY feature as library? Video conversion core, file operations, UI components
- Libraries listed: video-converter (FFmpeg wrapper), file-manager (drag-drop, validation), ui-components (shadcn-based)
- CLI per library: N/A (desktop GUI application)
- Library docs: README.md format for each module

Testing (NON-NEGOTIABLE):

- RED-GREEN-Refactor cycle enforced? ☑ Yes, tests written first
- Git commits show tests before implementation? ☑ Will be enforced
- Order: Contract→Integration→E2E→Unit strictly followed? ☑ Yes
- Real dependencies used? ☑ Real file system, actual FFmpeg binaries
- Integration tests for: file operations, video conversion, UI interactions
- FORBIDDEN: Implementation before test, skipping RED phase

Observability:

- Structured logging included? ☑ electron-log with structured output
- Frontend logs → backend? ☑ IPC logging bridge main↔renderer
- Error context sufficient? ☑ Full error context with stack traces

Versioning:

- Version number assigned? 1.0.0 (MAJOR.MINOR.BUILD)
- BUILD increments on every change? ☑ Yes
- Breaking changes handled? ☑ Semantic versioning for internal APIs

Project Structure

Documentation (this feature)

```
specs/001-desktop-video-converter/
├── plan.md              # This file (/plan command output)
├── research.md          # Phase 0 output (/plan command)
├── data-model.md        # Phase 1 output (/plan command)
├── quickstart.md        # Phase 1 output (/plan command)
├── contracts/           # Phase 1 output (/plan command)
```

```
└── tasks.md             # Phase 2 output (/tasks command - NOT created by
/plan)
```

Source Code (repository root)

```
# Electron Desktop Application Structure
src/
├── main/                # Electron main process
│   ├── index.ts         # Main entry point
│   ├── window.ts        # Window management
│   ├── ipc/             # IPC handlers
│   └── services/        # Core services (video conversion, file ops)
├── renderer/            # React frontend
│   ├── src/
│   │   ├── components/  # UI components (shadcn-based)
│   │   ├── pages/       # Application screens
│   │   ├── hooks/       # React hooks
│   │   ├── lib/         # Utilities
│   │   └── types/       # TypeScript types
│   ├── index.html
│   └── vite.config.ts
├── preload/             # Preload scripts
│   └── index.ts
└── shared/              # Shared types and utilities
    ├── types/
    └── constants/

tests/
├── e2e/                 # Playwright E2E tests
├── integration/         # Cross-process integration tests
└── unit/                # Unit tests (main + renderer)

build/                   # Build configuration
├── electron-builder.js
└── vite/
    ├── main.config.ts
    └── renderer.config.ts
```

Structure Decision: Desktop application with main/renderer/preload separation following Electron security best practices

Phase 0: Outline & Research

1. **Extract unknowns from Technical Context**:

    - Best practices for React + Electron + Vite integration
    - FFmpeg integration patterns for Electron applications
    - shadcn/ui setup and configuration for Electron renderer

   - Electron security best practices for file operations
   - Video format detection and validation approaches
   - Progress tracking patterns for long-running operations
   - Packaging strategies for single executable distribution

2. **Generate and dispatch research agents**:

   ```
   Task: "Research React + Electron + Vite integration best practices"
   Task: "Find optimal FFmpeg integration for Electron desktop apps"
   Task: "Research shadcn/ui setup for Electron renderer process"
   Task: "Investigate Electron security patterns for file operations"
   Task: "Research video format detection libraries"
   Task: "Find progress tracking patterns for video conversion"
   Task: "Research single executable packaging with electron-builder"
   ```

3. **Consolidate findings** in `research.md`

Output: research.md with all technical decisions documented

Phase 1: Design & Contracts

Prerequisites: research.md complete

1. **Extract entities from feature spec** → `data-model.md`:

   - VideoFile entity (path, name, format, size, validation status)
   - ConversionJob entity (progress, status, source, destination, settings)
   - ApplicationState entity (UI state, user preferences)

2. **Generate IPC contracts** from functional requirements:

   - File selection and validation APIs
   - Video conversion control APIs (start, cancel, progress)
   - File system operations APIs
   - Output to `/contracts/ipc-contracts.ts`

3. **Generate contract tests** from contracts:

   - IPC communication tests
   - File operation tests
   - Video conversion integration tests

4. **Extract test scenarios** from user stories:

   - File selection and drag-drop scenarios
   - Conversion progress and cancellation scenarios
   - Success and error handling scenarios

5. **Update agent file incrementally**:

  - Create `.github/copilot-instructions.md` for GitHub Copilot
  - Include Electron + React + Vite patterns
  - Include shadcn/ui component usage
  - Keep under 150 lines for token efficiency

Output: data-model.md, /contracts/*, failing tests, quickstart.md, .github/copilot-instructions.md

Phase 2: Task Planning Approach

This section describes what the /tasks command will do - DO NOT execute during /plan

Task Generation Strategy:

- Load `/templates/tasks-template.md` as base
- Generate tasks from Phase 1 design docs
- Each IPC contract → contract test task [P]
- Each entity → model creation task [P]
- Each user story → integration test task
- UI component tasks → shadcn implementation
- Electron packaging tasks → executable generation

Ordering Strategy:

- TDD order: Tests before implementation
- Dependency order: Shared types → Main process → Preload → Renderer
- Mark [P] for parallel execution (independent modules)

Estimated Output: 35-40 numbered, ordered tasks in tasks.md

Progress Tracking

- [x] Feature spec loaded
- [x] Technical context filled
- [x] Initial constitution check
- [ ] Phase 0: Research execution
- [ ] Phase 1: Design & contracts
- [ ] Post-design constitution check
- [ ] Phase 2: Task planning approach