Technical Specifications

StreamGO

1. INTRODUCTION

1.1 EXECUTIVE SUMMARY

1.1.1 Brief Overview of the Project

The Stremio Clone Desktop Application project aims to develop a comprehensive, cross-platform media center application that replicates and enhances the core functionality of Stremio, a popular open-source streaming platform. This application will be built using Tauri 2.0, a framework for building tiny and fast binaries for all major desktop platforms, allowing developers to integrate any frontend framework that compiles to HTML, JavaScript, and CSS while leveraging Rust for backend logic.

The application will serve as a unified media aggregation platform, enabling users to discover, organize, and stream content from multiple sources through an extensible addon system. By combining modern web technologies with Rust's performance and security advantages, the solution will deliver a lightweight, secure, and highly performant desktop experience across Windows, macOS, and Linux platforms.

1.1.2 Core Business Problem Being Solved

Over 90% of U.S. adults use streaming platforms, while only 40% still have cable or satellite TV, highlighting the massive shift towards on-demand content consumption. However, the current streaming landscape presents several critical challenges:

- **Content Fragmentation**: Users must subscribe to multiple streaming services to access desired content, leading to subscription fatigue and increased costs

- **Platform Lock-in**: Content is siloed across different proprietary platforms with inconsistent user experiences
- **Limited Customization**: Existing solutions offer minimal personalization and extensibility options
- **Performance Issues**: Many media center applications are slower due to feature-rich plugins that often slow down basic functions

The Stremio Clone addresses these pain points by providing a unified, customizable, and high-performance media center that aggregates content from multiple sources while maintaining user privacy and system security.

1.1.3 Key Stakeholders and Users

Stakeholder Group	Primary Interests	Usage Patterns
End Users (Cord-Cutters)	Unified content access, cost reduction, seamless experience	Daily streaming, content discovery, library management
Content Enthusiasts	Extensive customization, addon ecosystem, advanced features	Heavy usage, community participation, addon development
Privacy-Conscious Users	Data protection, local storage, secure streaming	Selective usage, security-focused configurations

1.1.4 Expected Business Impact and Value Proposition

Primary Value Propositions:

- **Unified Experience**: Single interface for accessing content from multiple streaming platforms and sources
- **Cost Efficiency**: Reduces need for multiple streaming subscriptions through content aggregation

- **Performance Excellence**: Streamlined design that operates much faster than leading apps like Kodi, especially on PC
- **Enhanced Privacy**: Local data storage and processing with minimal external dependencies
- **Extensibility**: Robust addon system enabling community-driven feature expansion

Expected Market Impact:
The global media streaming market is expected to reach USD 285.4 billion by 2034, growing at a CAGR of 10.6%, indicating substantial opportunity for innovative streaming solutions that address current market limitations.

1.2 SYSTEM OVERVIEW

1.2.1 Project Context

Business Context and Market Positioning

Stremio is a media center app that has taken the streaming world by storm in recent years, with its sleek interface, broad content library, and powerful add-on capabilities providing an unparalleled viewing experience. The market positioning focuses on competing with established media center solutions while offering superior performance and user experience.

Competitive Landscape:

- **Primary Competitors**: Netflix (market leader), Kodi (open-source alternative), Jellyfin (self-hosted solution)
- **Differentiation**: Stremio sets itself apart by keeping things simple and smooth, with performance much better than leading apps like Kodi

Current System Limitations

Existing media center solutions face several critical limitations:

- **Complexity**: Kodi offers greater customization through its extensive addon library, though this comes with a steeper learning curve
- **Security Concerns**: Traditional solutions install addons directly on user computers, creating potential security risks from malicious developers
- **Performance Issues**: Resource-intensive operations and slow loading times
- **Platform Fragmentation**: Inconsistent experiences across different operating systems

Integration with Existing Enterprise Landscape

The application will integrate with existing digital ecosystems through:

- **Streaming Service APIs**: Direct integration with legitimate streaming platforms
- **Media Database Services**: TMDB for metadata and content information
- **Cloud Storage**: Optional synchronization with cloud services for user data
- **Casting Protocols**: Support for Chromecast, DLNA, and AirPlay

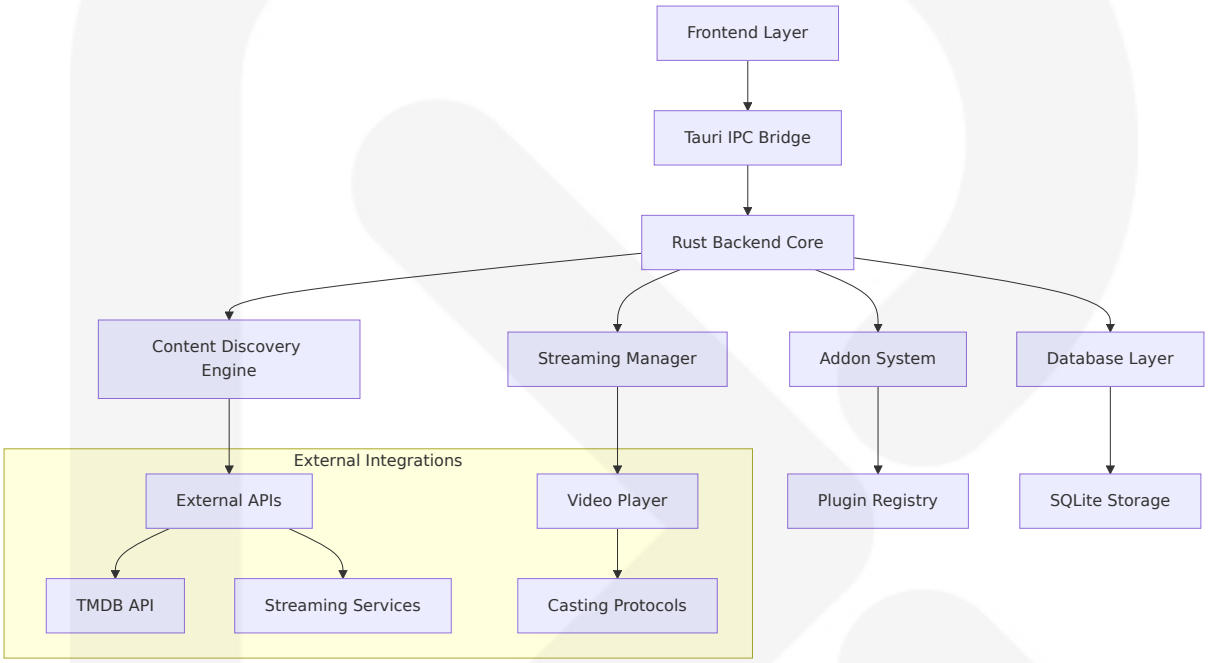
1.2.2 High-Level Description

Primary System Capabilities

Capability Category	Core Functions
Content Discovery	Search, browse, recommendation engine, metadata aggregation
Streaming Playback	Multi-format video player, subtitle support, quality selection
Library Management	Watchlist, progress tracking, favorites, offline content

Capability Category	Core Functions
Addon System	Plugin installation, configuration, community marketplace

Major System Components



Core Technical Approach

The frontend will be written in web technologies and run inside the operating system WebView, communicating with the application core written in Rust. This architecture provides:

- **Security**: Remote addon processing on servers rather than local execution, significantly reducing security risks by preventing malicious developers from directly accessing the operating system
- **Performance**: Native Rust backend for computationally intensive operations
- **Cross-Platform**: Unified interface leveraging system webviews (WKWebView on macOS, WebView2 on Windows, WebKitGTK on Linux)

1.2.3 Success Criteria

Measurable Objectives

Metric Category	Target Value	Measurement Method
Performance	Application startup < 3 seconds	Automated performance testing
User Adoption	10,000+ active users within 6 months	Analytics tracking
Content Coverage	95% successful content discovery	Content availability testing
System Stability	99.5% uptime, < 0.1% crash rate	Error monitoring and reporting

Critical Success Factors

- **User Experience**: Intuitive interface with minimal learning curve
- **Content Availability**: Reliable access to diverse content sources
- **Performance**: Consistent high-speed operation across all supported platforms
- **Security**: Robust protection against malicious addons and data breaches
- **Community Adoption**: Active addon development and user contribution ecosystem

Key Performance Indicators (KPIs)

- **Technical KPIs**: Response time, memory usage, CPU utilization, error rates
- **User Engagement KPIs**: Daily active users, session duration, content consumption patterns
- **Business KPIs**: User retention rate, addon installation frequency, community growth metrics

1.3 SCOPE

1.3.1 In-Scope

Core Features and Functionalities

Feature Category	Included Capabilities
User Management	Authentication, profile syncing, watch history, favorites
Content Discovery	Search functionality, browse interface, metadata integration
Streaming Playback	Video player integration, subtitle support, quality selection
Library Organization	Watchlist management, progress tracking, recommendations

Implementation Boundaries

- **Supported Platforms**: Windows 10+, macOS 10.15+, Linux (Ubuntu 22.04+)
- **Content Sources**: Public APIs, legitimate streaming services, user-provided content
- **User Base**: Individual consumers and small household deployments
- **Geographic Coverage**: Global deployment with localization support for major markets

Key Technical Requirements

- **Framework**: Tauri 2.0 stable release with mobile support capabilities
- **Backend Language**: Rust with async/await support using Tokio runtime
- **Frontend Technologies**: HTML5, CSS3, JavaScript (framework-agnostic)

- **Database**: SQLite for local storage with optional cloud synchronization
- **Security**: Input sanitization, secure IPC, encrypted local storage

1.3.2 Out-of-Scope

Explicitly Excluded Features/Capabilities

- **Mobile Applications**: While Tauri 2.0 supports iOS and Android, mobile versions are excluded from initial release
- **Content Hosting**: No direct content storage or distribution capabilities
- **Live TV Broadcasting**: Excludes traditional broadcast television integration
- **Enterprise Features**: Multi-tenant architecture, advanced user management, enterprise SSO

Future Phase Considerations

- **Phase 2**: Mobile application development using Tauri's mobile capabilities
- **Phase 3**: Smart TV and embedded device support
- **Phase 4**: Enterprise features and advanced analytics
- **Phase 5**: AI-powered content recommendation engine

Integration Points Not Covered

- **Social Media Integration**: Sharing capabilities and social features
- **Payment Processing**: Subscription management and billing systems
- **Advanced Analytics**: User behavior tracking and business intelligence
- **Content Creation Tools**: Video editing or content production features

Unsupported Use Cases

- **Commercial Distribution**: Large-scale content distribution or CDN functionality
- **Live Streaming Production**: Broadcasting or streaming content creation
- **Enterprise Deployment**: Multi-user enterprise installations with centralized management
- **Offline-First Architecture**: Complete offline functionality without internet connectivity

2. PRODUCT REQUIREMENTS

2.1 FEATURE CATALOG

2.1.1 Core Authentication and User Management

Feature ID	F-001
Feature Name	User Authentication System
Category	Core Infrastructure
Priority	Critical
Status	Proposed

Description:

- **Overview**: Tauri 2.0 supports all major desktop (macOS, linux, windows) and mobile (iOS, Android) platforms, allowing developers to integrate any frontend framework that compiles to HTML, JavaScript, and CSS for building their user experience while leveraging languages such as Rust, Swift, and Kotlin for backend logic when needed. The

authentication system provides secure user account management with local and cloud synchronization capabilities.
- **Business Value**: Enables personalized user experiences, watch history tracking, and cross-device synchronization
- **User Benefits**: Seamless access across devices, personalized recommendations, and secure data storage
- **Technical Context**: Implements secure authentication using Rust backend with SQLite local storage and optional cloud sync

Dependencies:

- **Prerequisite Features**: None (foundational feature)
- **System Dependencies**: SQLite database, Tauri IPC system
- **External Dependencies**: Optional cloud authentication service
- **Integration Requirements**: Secure storage for user credentials and session management

Feature ID	F-002
Feature Name	Content Discovery Engine
Category	Core Functionality
Priority	Critical
Status	Proposed

Description:

- **Overview**: TMDB provides the definitive list of currently available methods for movie, tv, actor and image API, along with extensive metadata for movies, TV shows and people, and one of the best selections of high resolution posters and backdrops. Comprehensive search and browse functionality for discovering movies, TV shows, and other media content.
- **Business Value**: Core functionality that enables users to find and access content efficiently

- **User Benefits**: Fast, accurate content discovery with rich metadata and visual assets
- **Technical Context**: Integrates with TMDB API for metadata, implements advanced search algorithms

Dependencies:

- **Prerequisite Features**: F-001 (User Authentication System)
- **System Dependencies**: HTTP client for API requests, caching system
- **External Dependencies**: TMDB API (free for non-commercial use with attribution)
- **Integration Requirements**: API key management, rate limiting, offline caching

Feature ID	F-003
Feature Name	Addon System Architecture
Category	Core Infrastructure
Priority	Critical
Status	Proposed

Description:

- **Overview**: An add-on in Stremio doesn't generally run on a client's computer. Instead, it is hosted on the Internet just like any website. This brings ease of use and security benefits to the end user. Extensible plugin system for integrating content sources and additional functionality.
- **Business Value**: Enables ecosystem growth through community contributions and third-party integrations
- **User Benefits**: Access to diverse content sources, customizable functionality, enhanced security
- **Technical Context**: The plugins usually do not depend on other plugins, with some exceptions. This means to implement a new file system access functionality it is only required to contribute to the fs

plugin instead of Tauri itself. As this release also targets mobile platforms, the plugin system also supports mobile plugins

Dependencies:

- **Prerequisite Features**: F-001 (User Authentication System)
- **System Dependencies**: Tauri plugin system, HTTP client for remote addon communication
- **External Dependencies**: Remote addon servers, addon manifest validation
- **Integration Requirements**: Secure addon loading, permission management, CORS handling

2.1.2 Media Playback and Streaming

Feature ID	F-004
Feature Name	Video Player Integration
Category	Media Playback
Priority	Critical
Status	Proposed

Description:

- **Overview**: High-performance video player with support for multiple formats, quality selection, and subtitle management
- **Business Value**: Core streaming functionality that enables content consumption
- **User Benefits**: Smooth playback experience, format compatibility, accessibility features
- **Technical Context**: Web-based player integration with Tauri's webview system

Dependencies:

- **Prerequisite Features**: F-002 (Content Discovery Engine), F-003 (Addon System)
- **System Dependencies**: Media codecs, hardware acceleration support
- **External Dependencies**: Video.js or similar web player library
- **Integration Requirements**: Stream URL handling, subtitle file support, casting protocols

Feature ID	F-005
Feature Name	Casting and External Device Support
Category	Media Playback
Priority	High
Status	Proposed

Description:

- **Overview**: Support for casting content to external devices including Chromecast, DLNA, and AirPlay
- **Business Value**: Extends viewing experience beyond desktop to TV and other devices
- **User Benefits**: Flexible viewing options, enhanced user experience
- **Technical Context**: Implements casting protocols through Tauri's system integration capabilities

Dependencies:

- **Prerequisite Features**: F-004 (Video Player Integration)
- **System Dependencies**: Network discovery, casting protocol libraries
- **External Dependencies**: Device-specific casting SDKs
- **Integration Requirements**: Network permissions, device discovery protocols

2.1.3 Library Management and Organization

Feature ID	F-006
Feature Name	Personal Library Management
Category	Content Organization
Priority	High
Status	Proposed

Description:

- **Overview**: Comprehensive library system for organizing watchlists, tracking progress, and managing favorites
- **Business Value**: Enhances user engagement through personalized content organization
- **User Benefits**: Organized content access, progress tracking, personalized recommendations
- **Technical Context**: Local database storage with cloud synchronization capabilities

Dependencies:

- **Prerequisite Features**: F-001 (User Authentication System), F-002 (Content Discovery Engine)
- **System Dependencies**: SQLite database, synchronization service
- **External Dependencies**: Optional cloud storage service
- **Integration Requirements**: Data synchronization, conflict resolution, offline access

Feature ID	F-007
Feature Name	Offline Content Management
Category	Content Organization
Priority	Medium
Status	Proposed

Description:

- **Overview**: System for caching metadata and enabling offline content access where legally permitted
- **Business Value**: Improves user experience in low-connectivity scenarios
- **User Benefits**: Continued access to library and cached content without internet
- **Technical Context**: Intelligent caching system with storage management

Dependencies:

- **Prerequisite Features**: F-006 (Personal Library Management)
- **System Dependencies**: Local file system, storage management
- **External Dependencies**: Content licensing permissions
- **Integration Requirements**: Cache invalidation, storage quotas, legal compliance

2.1.4 User Interface and Experience

Feature ID	F-008
Feature Name	Modern Responsive Interface
Category	User Interface
Priority	High
Status	Proposed

Description:

- **Overview**: Modern, intuitive interface with sections for home/discover, library, search, and settings. The frontend is written in web technologies and runs inside the operating system WebView, communicating with the application core written mostly in Rust
- **Business Value**: Provides competitive user experience that drives adoption and retention

- **User Benefits**: Intuitive navigation, responsive design, themeable interface
- **Technical Context**: Web-based UI leveraging system webviews for native performance

Dependencies:

- **Prerequisite Features**: All core features (F-001 through F-007)
- **System Dependencies**: System WebView (Chromium on Windows, Safari on macOS, WebKitGTK on Linux)
- **External Dependencies**: Frontend framework (React, Vue, or Svelte)
- **Integration Requirements**: Tauri IPC for backend communication, responsive design principles

2.2 FUNCTIONAL REQUIREMENTS TABLE

2.2.1 User Authentication System (F-001)

Requirement ID	F-001-RQ-001
Description	User account creation and login functionality
Acceptance Criteria	- Users can create accounts with email/password - Secure password hashing and storage - Session management with automatic logout - Password reset functionality
Priority	Must-Have
Complexity	Medium

Technical Specifications:

- **Input Parameters**: Email, password, optional profile information
- **Output/Response**: Authentication token, user profile data

- **Performance Criteria**: Login response time < 2 seconds
- **Data Requirements**: Encrypted user credentials, session tokens

Validation Rules:

- **Business Rules**: Unique email addresses, password complexity requirements
- **Data Validation**: Email format validation, password strength checks
- **Security Requirements**: Bcrypt password hashing, secure session storage
- **Compliance Requirements**: GDPR compliance for user data handling

Requirement ID	F-001-RQ-002
Description	Cross-device synchronization of user data
Acceptance Criteria	- Watch history syncs across devices - Library preferences maintained - Settings synchronized - Conflict resolution for simultaneous updates
Priority	Should-Have
Complexity	High

Technical Specifications:

- **Input Parameters**: User authentication token, device identifier
- **Output/Response**: Synchronized user data, sync status
- **Performance Criteria**: Sync completion within 10 seconds
- **Data Requirements**: Timestamped data changes, conflict resolution metadata

2.2.2 Content Discovery Engine (F-002)

Requirement ID	F-002-RQ-001
Description	Advanced search functionality with filters

Requirement ID	F-002-RQ-001
Acceptance Criteria	- Text-based search across titles, actors, genres - Filter by year, rating, genre, type - Sort by relevance, popularity, date - Search suggestions and autocomplete
Priority	Must-Have
Complexity	Medium

Technical Specifications:

- **Input Parameters**: Search query, filter criteria, sort preferences
- **Output/Response**: Paginated search results with standard movie list objects
- **Performance Criteria**: Search response time < 1 second
- **Data Requirements**: Extensive metadata for movies, TV shows and people, with over 1,000 images added daily

Validation Rules:

- **Business Rules**: Minimum search query length, result pagination limits
- **Data Validation**: Input sanitization, query parameter validation
- **Security Requirements**: Rate limiting, input sanitization
- **Compliance Requirements**: TMDB attribution: "This product uses the TMDB API but is not endorsed or certified by TMDB."

Requirement ID	F-002-RQ-002
Description	Content metadata aggregation and display
Acceptance Criteria	- Rich metadata display (plot, cast, ratings) - High-quality poster and backdrop images - Trailer and video content integration - Multi-language support for metadata
Priority	Must-Have
Complexity	Medium

Technical Specifications:

- **Input Parameters**: Content ID, language preference
- **Output/Response**: Complete movie details with append_to_response support
- **Performance Criteria**: Metadata loading time < 3 seconds
- **Data Requirements**: Support for 39 languages with extensive regional data

2.2.3 Addon System Architecture (F-003)

Requirement ID	F-003-RQ-001
Description	Remote addon discovery and installation
Acceptance Criteria	- Browse available addons from repositories - Install addons with one-click - Addon configuration interface - Automatic addon updates
Priority	Must-Have
Complexity	High

Technical Specifications:

- **Input Parameters**: Addon URL, configuration parameters
- **Output/Response**: Addon manifest, installation status
- **Performance Criteria**: Addon installation time < 30 seconds
- **Data Requirements**: Addon manifest JSON describing capabilities and resources

Validation Rules:

- **Business Rules**: Addon URLs must be loaded with HTTPS (except 127.0.0.1) and must support CORS
- **Data Validation**: Manifest schema validation, URL format checking
- **Security Requirements**: Addons do not run any code locally, so they pose no risks to your device

- **Compliance Requirements**: Addon content policy compliance

Requirement ID	F-003-RQ-002
Description	Secure addon communication and content delivery
Acceptance Criteria	- HTTPS-only addon communication - Content stream URL validation - Addon permission management - Error handling for failed addon requests
Priority	Must-Have
Complexity	High

Technical Specifications:

- **Input Parameters**: Addon endpoint, request parameters
- **Output/Response**: Content streams, metadata, error codes
- **Performance Criteria**: Addon response time < 5 seconds
- **Data Requirements**: Addon resources accessed at specific endpoints with proper data response

2.2.4 Video Player Integration (F-004)

Requirement ID	F-004-RQ-001
Description	Multi-format video playback support
Acceptance Criteria	- Support for MP4, WebM, HLS, DASH formats - Adaptive quality selection - Subtitle support (SRT, VTT, ASS) - Playback controls (play, pause, seek, volume)
Priority	Must-Have
Complexity	Medium

Technical Specifications:

- **Input Parameters**: Stream URL, subtitle files, quality preferences

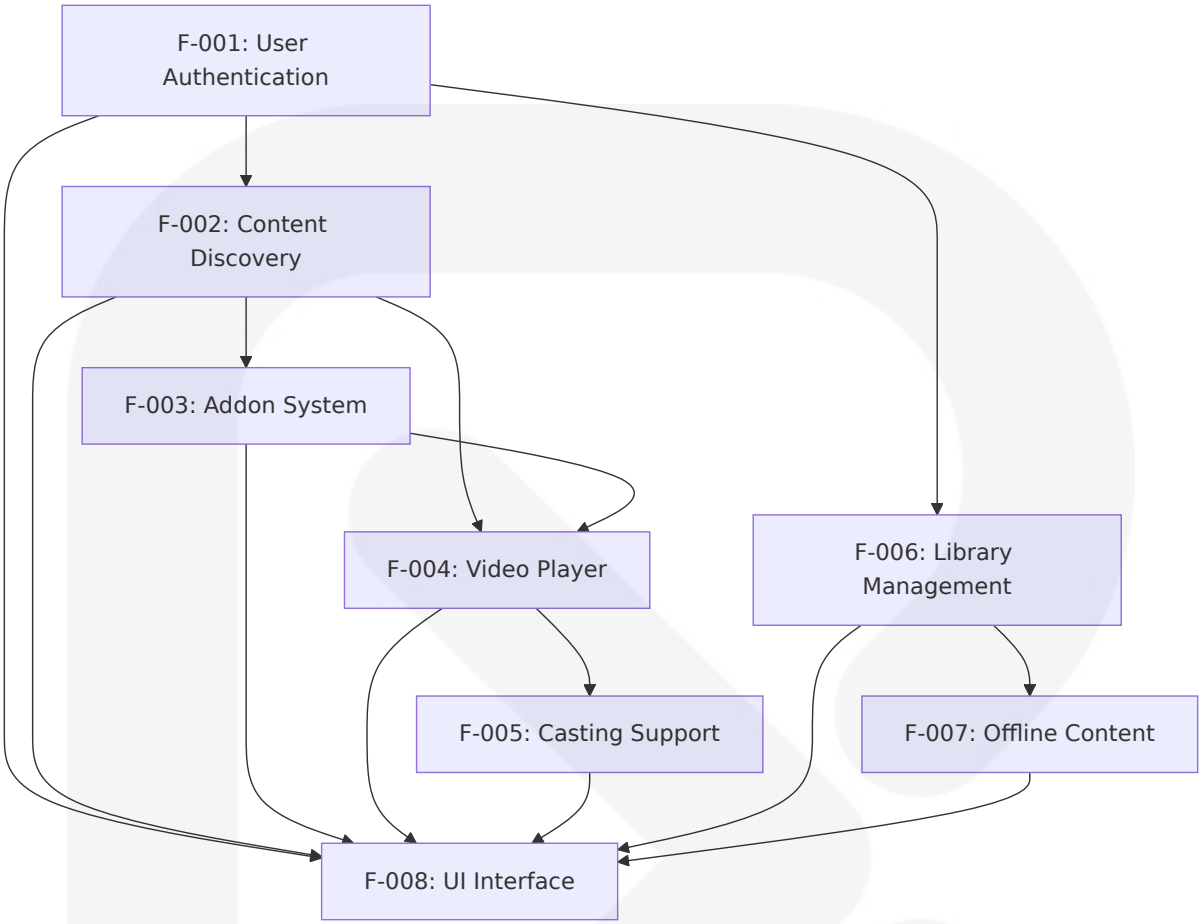
- **Output/Response**: Video playback status, current position, quality level
- **Performance Criteria**: Video start time < 5 seconds, smooth playback
- **Data Requirements**: Video stream metadata, subtitle timing data

Validation Rules:

- **Business Rules**: Supported format validation, quality level limits
- **Data Validation**: Stream URL validation, subtitle format checking
- **Security Requirements**: Stream URL sanitization, secure content delivery
- **Compliance Requirements**: Content licensing verification

2.3 FEATURE RELATIONSHIPS

2.3.1 Feature Dependencies Map



2.3.2 Integration Points

Integration Point	Connected Features	Shared Components
User Data Sync	F-001, F-006, F-007	SQLite database, cloud sync service
Content Metadata	F-002, F-006, F-008	TMDB API client, caching layer
Stream Processing	F-003, F-004, F-005	HTTP client, stream validator
UI State Management	F-001, F-002, F-006, F-008	Frontend state store, IPC bridge

2.3.3 Common Services

Service Name	Supporting Features	Technical Implementation
Database Service	F-001, F-002, F-006, F-007	SQLite with Rust ORM (diesel/sqlx)
HTTP Client Service	F-002, F-003, F-004	reqwest crate with connection pooling
Cache Management	F-002, F-007, F-008	In-memory and disk-based caching
IPC Communication	All Features	Tauri command system for secure frontend-backend communication

2.4 IMPLEMENTATION CONSIDERATIONS

2.4.1 Technical Constraints

Feature	Constraints	Mitigation Strategy
F-001	WebView security limitations	Use Tauri's secure IPC for sensitive operations
F-002	TMDB API rate limits, no SLA provided	Implement caching and request queuing
F-003	HTTPS requirement for remote addons	Enforce HTTPS validation, provide development tools
F-004	WebView codec support varies by platform	Use web-compatible formats, fallback options

2.4.2 Performance Requirements

Feature Category	Performance Target	Measurement Method
Authentication	Login response < 2 seconds	Automated performance testing
Content Discovery	Search results < 1 second	Response time monitoring
Video Playback	Stream start < 5 seconds	Playback analytics
UI Responsiveness	Interaction response < 100ms	Frontend performance profiling

2.4.3 Security Implications

Security Aspect	Implementation Approach	Validation Method
User Data Protection	Encrypted local storage, secure transmission	Security audit, penetration testing
Addon Security	Remote execution model prevents local code execution	Addon manifest validation
Content Stream Validation	URL sanitization, HTTPS enforcement	Automated security scanning
Cross-Origin Security	CORS header validation for addon communication	Browser security testing

2.4.4 Scalability Considerations

Scalability Factor	Design Approach	Monitoring Strategy
User Growth	Stateless authentication, local data storage	User metrics tracking
Content Catalog Size	Efficient search indexing, pagination	Query performance monitoring

Scalability Factor	Design Approach	Monitoring Strategy
Addon Ecosystem	Distributed addon architecture	Addon performance analytics
Cross-Platform Support	Unified codebase for Windows, macOS, Linux platforms	Platform-specific testing

2.4.5 Maintenance Requirements

Maintenance Area	Requirements	Automation Level
Dependency Updates	Regular Tauri 2.0 stable updates	Automated dependency checking
API Integration	TMDB API changes, addon compatibility	Automated API testing
Security Patches	Regular security updates, vulnerability scanning	Automated security monitoring
Platform Compatibility	OS update compatibility testing	Continuous integration testing

3. TECHNOLOGY STACK

3.1 PROGRAMMING LANGUAGES

3.1.1 Backend Development

Language	Version	Platform	Justification
Rust	1.70+ (MSRV)	All Platforms	Tauri is a framework for building tiny and fast binaries for all major desktop (macOS, linux, windows) and mobile (iOS, Android) platforms. De

Language	Version	Platform	Justification
			velopers can integrate any frontend framework that compiles to HTML, JavaScript, and CSS for building their user experience while leveraging languages such as Rust, Swift, and Kotlin for backend logic when needed. Provides memory safety, performance, and cross-platform compatibility.

Selection Criteria:

- **Memory Safety**: Rust's ownership system prevents common security vulnerabilities like buffer overflows and memory leaks
- **Performance**: Zero-cost abstractions and compile-time optimizations deliver native performance
- **Ecosystem**: Rich crate ecosystem with mature libraries for HTTP clients, database access, and async programming
- **Tauri Integration**: In a Tauri application the frontend is written in your favorite web frontend stack. This runs inside the operating system WebView and communicates with the application core written mostly in Rust.

3.1.2 Frontend Development

Language	Version	Platform	Justification
JavaScript	ES2022+	WebView	Standard web technology for dynamic UI interactions and API communication
HTML5	Latest	WebView	Semantic markup for application structure and accessibility
CSS3	Latest	WebView	Modern styling with flexbox, grid, and custom properties support

Selection Criteria:

- **Framework Agnostic**: Tauri supports any frontend framework so you don't need to change your stack.
- **WebView Compatibility**: Leverages system WebViews for native performance and reduced bundle size
- **Development Flexibility**: Allows choice of modern frontend frameworks (React, Vue, Svelte) without architectural constraints

3.1.3 Platform-Specific Extensions

Language	Version	Platform	Use Case
Swift	5.0+	iOS/macOS	You can write or re-use native code in Swift on iOS and Kotlin on Android and directly expose functions to the Tauri frontend using Annotations (@Command on Android), implementing a Subclass (YourPluginClass: Plugin) on iOS, or by invoking the Swift or Kotlin code from a Rust based Tauri command.
Kotlin	1.8+	Android	Native Android integrations and platform-specific functionality

3.2 FRAMEWORKS & LIBRARIES

3.2.1 Core Application Framework

Framework	Version	Purpose	Justification
Tauri	2.0 Stable	Cross-platform desktop application framework	We are very proud to finally announce the stable release for the new major version of Tauri. Welcome to Tauri 2.0! Provides secure, lightweight desktop ap

Framework	Version	Purpose	Justification
			plications with web technologies.

Key Features:

- We hope to stabilize the core functionality and offer a stable framework, where the moving parts are mostly plugins offering access to system specific functionality. You no longer need to understand all of Tauri to improve or implement specific features.
- The plugins usually do not depend on other plugins, with some exceptions. This means to implement a new file system access functionality it is only required to contribute to the fs plugin instead of Tauri itself.
- As this release also targets mobile platforms, the plugin system also supports mobile plugins.

3.2.2 Async Runtime

Framework	Version	Purpose	Justification
Tokio	1.47+	Asynchronous runtime for Rust	Tokio is a runtime for writing reliable asynchronous applications with Rust. It provides async I/O, networking, scheduling, timers, and more. Essential for handling concurrent HTTP requests and I/O operations.

Runtime Configuration:

- Our current LTS releases are: 1.43.x - LTS release until March 2026. (MSRV 1.70) 1.47.x - LTS release until September 2026.
- Multi-threaded runtime for CPU-intensive operations
- Current thread runtime for lightweight tasks

3.2.3 Frontend Framework Options

Framework	Version	Compatibility	Recommendation
React	18+	Full	Recommended for complex state management
Vue.js	3+	Full	Recommended for rapid development
Svelte	4+	Full	Recommended for minimal bundle size
Vanilla JS	ES2022+	Full	Recommended for maximum performance

Selection Criteria:

- WebView compatibility across all target platforms
- Bundle size optimization for desktop applications
- Development team expertise and preferences
- Component ecosystem availability

3.3 OPEN SOURCE DEPENDENCIES

3.3.1 HTTP Client Libraries

Crate	Version	Purpose	Features
reqwest	0.12+	HTTP client library	An ergonomic, batteries-included HTTP Client for Rust. JSON support, async/blocking APIs, TLS, cookies
serde	1.0+	Serialization framework	JSON/YAML parsing, derive macros
serde_json	1.0+	JSON serialization	High-performance JSON processing

HTTP Client Configuration:

```
[dependencies]
reqwest = { version = "0.12", features = ["json", "rustls-tls"] }
```

3.3.2 Database Libraries

Crate	Version	Purpose	Features
sqlx	0.7+	Async SQL toolkit	Compile-time checked queries, connection pooling
rusqlite	0.30+	SQLite bindings	Synchronous SQLite access, backup support

3.3.3 Utility Libraries

Crate	Version	Purpose	Features
anyhow	1.0+	Error handling	Context-aware error propagation
thiserror	1.0+	Error derive macros	Custom error types with derive support
uuid	1.0+	UUID generation	Unique identifier generation
chrono	0.4+	Date/time handling	Timezone-aware datetime operations

3.3.4 Tauri Plugin Ecosystem

Plugin	Version	Purpose	Platform Support
tauri-plugin-fs	2.0+	File system access	Desktop, Mobile
tauri-plugin-http	2.0+	HTTP requests	Desktop, Mobile

Plugin	Version	Purpose	Platform Support
tauri-plugin-shell	2.0+	System shell access	Desktop
tauri-plugin-window-state	2.0+	Window state persistence	Desktop

3.4 THIRD-PARTY SERVICES

3.4.1 Content Metadata Services

Service	API Version	Purpose	Usage Limits
TMDB API	v3	Movie/TV metadata and images	Welcome to version 3 of The Movie Database (TMDB) API. This is where you will find the definitive list of currently available methods for our movie, tv, actor and image API.

TMDB Integration Details:

- Our API is free to use for non-commercial purposes as long as you attribute TMDB as the source of the data and/or images.
- We do not currently provide an SLA. However, we do make every reasonable attempt to keep our service online and accessible.
- You shall place the following notice prominently on your application: "This product uses the TMDB API but is not endorsed or certified by TMDB."

3.4.2 Authentication Services

Service	Purpose	Integration Method
Local Authentication	User account management	SQLite-based credential storage
Optional Cloud Sync	Cross-device synchronization	RESTful API integration

3.4.3 Content Delivery

Service Type	Purpose	Implementation
Addon Repositories	Plugin distribution	HTTPS-based manifest delivery
Stream Validation	Content URL verification	Client-side URL sanitization

3.5 DATABASES & STORAGE

3.5.1 Primary Database

Database	Version	Purpose	Justification
SQLite	3.47+	Local data storage	SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world.

SQLite Features:

- The SQLite file format is stable, cross-platform, and backwards compatible and the developers pledge to keep it that way through the year 2050.
- In 2024, SQLite added support for JSONB, a binary serialization of SQLite's internal representation of JSON. Using JSONB allows

applications to avoid having to parse the JSON text each time it is processed and saves a small amount of disk space.
- Write-Ahead Logging (WAL) mode for improved concurrency
- Full-text search capabilities for content discovery

3.5.2 Data Persistence Strategy

Data Type	Storage Method	Synchronization
User Profiles	SQLite with encryption	Optional cloud backup
Watch History	Local SQLite database	Cross-device sync available
Content Metadata	SQLite with TTL caching	TMDB API refresh
Application Settings	SQLite configuration tables	Local persistence only

3.5.3 Caching Solutions

Cache Type	Implementation	Purpose
HTTP Response Cache	In-memory LRU cache	API response optimization
Image Cache	File system cache	Poster/backdrop storage
Metadata Cache	SQLite with expiration	Offline content access

3.6 DEVELOPMENT & DEPLOYMENT

3.6.1 Development Tools

Tool	Version	Purpose	Platform Support
Tauri CLI	2.8+	Application development and building	Windows, macOS, Linux
Rust Toolchain	1.70+	Backend compilation	Cross-platform

Tool	Version	Purpose	Platform Support
Node.js	18+	Frontend build tools	Development environment

3.6.2 Build System

Component	Technology	Configuration
Rust Backend	Cargo	Workspace-based multi-crate setup
Frontend Assets	Vite/Webpack	Modern bundling with tree-shaking
Cross-compilation	Cargo + Tauri	Automated multi-platform builds

3.6.3 Distribution Strategy

Platform	Package Format	Distribution Method
Windows	MSI/NSIS Installer	Direct download, Microsoft Store (future)
macOS	DMG/PKG	Direct download, App Store (future)
Linux	AppImage/DEB/RPM	Direct download, package repositories

3.6.4 CI/CD Requirements

Stage	Tools	Purpose
Testing	Cargo test, Jest	Unit and integration testing
Security Scanning	cargo-audit, SAST tools	Vulnerability detection
Cross-platform Building	GitHub Actions	Automated multi-platform releases
Code Quality	Clippy, ESLint	Static analysis and linting

3.6.5 Security Considerations

Security Layer	Implementation	Validation
IPC Security	Tauri command validation	Input sanitization and type checking
Content Security	HTTPS enforcement for addons	URL validation and CORS handling
Local Storage	SQLite encryption at rest	Encrypted user credentials and sensitive data
Update Mechanism	Signed update packages	Digital signature verification

3.6.6 Performance Monitoring

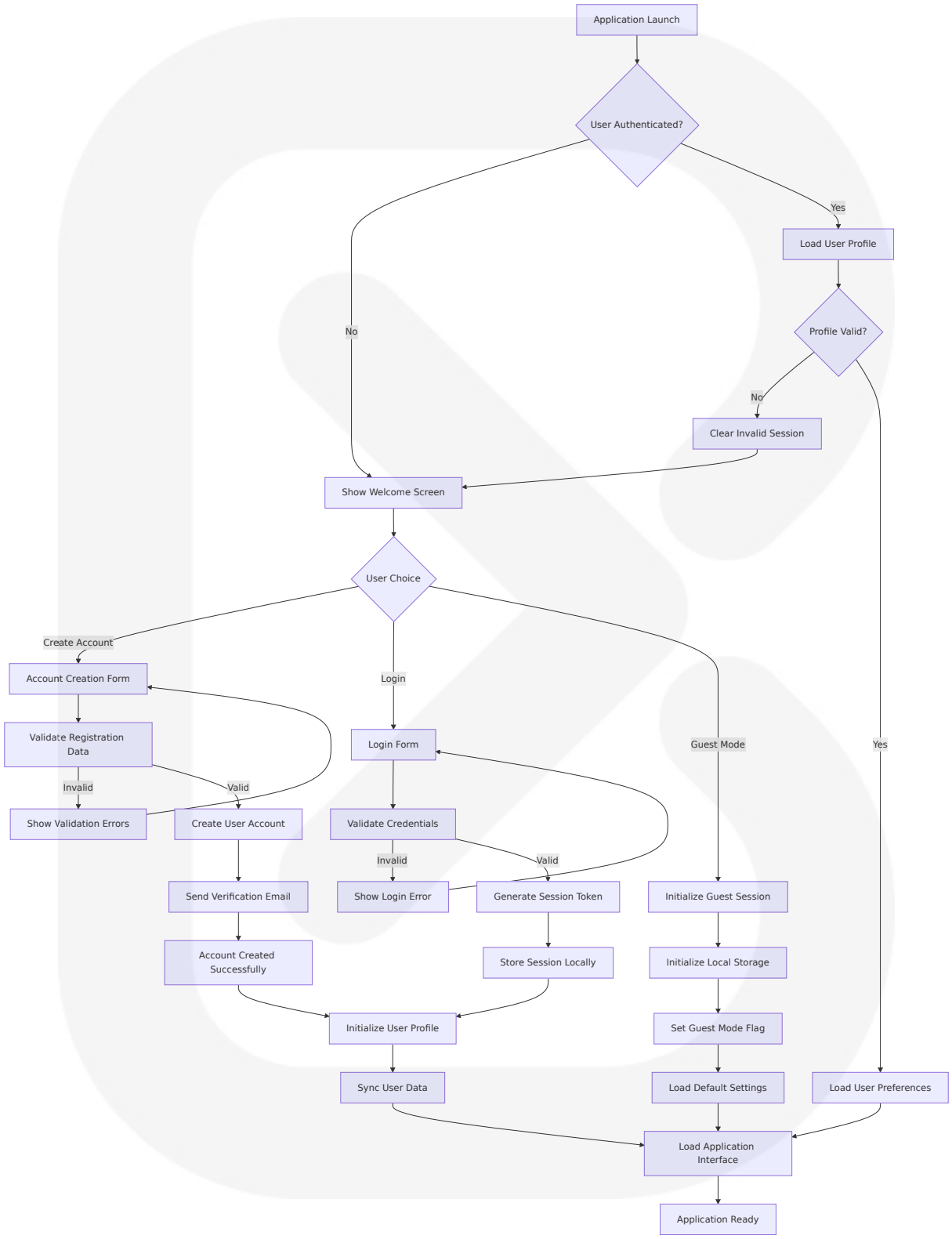
Metric Category	Monitoring Approach	Tools
Application Performance	Built-in profiling	Tokio console, custom metrics
Memory Usage	Runtime monitoring	System resource tracking
Network Performance	Request timing	HTTP client metrics
User Experience	Response time tracking	Frontend performance APIs

4. PROCESS FLOWCHART

4.1 SYSTEM WORKFLOWS

4.1.1 Core Business Processes

User Authentication and Onboarding Flow



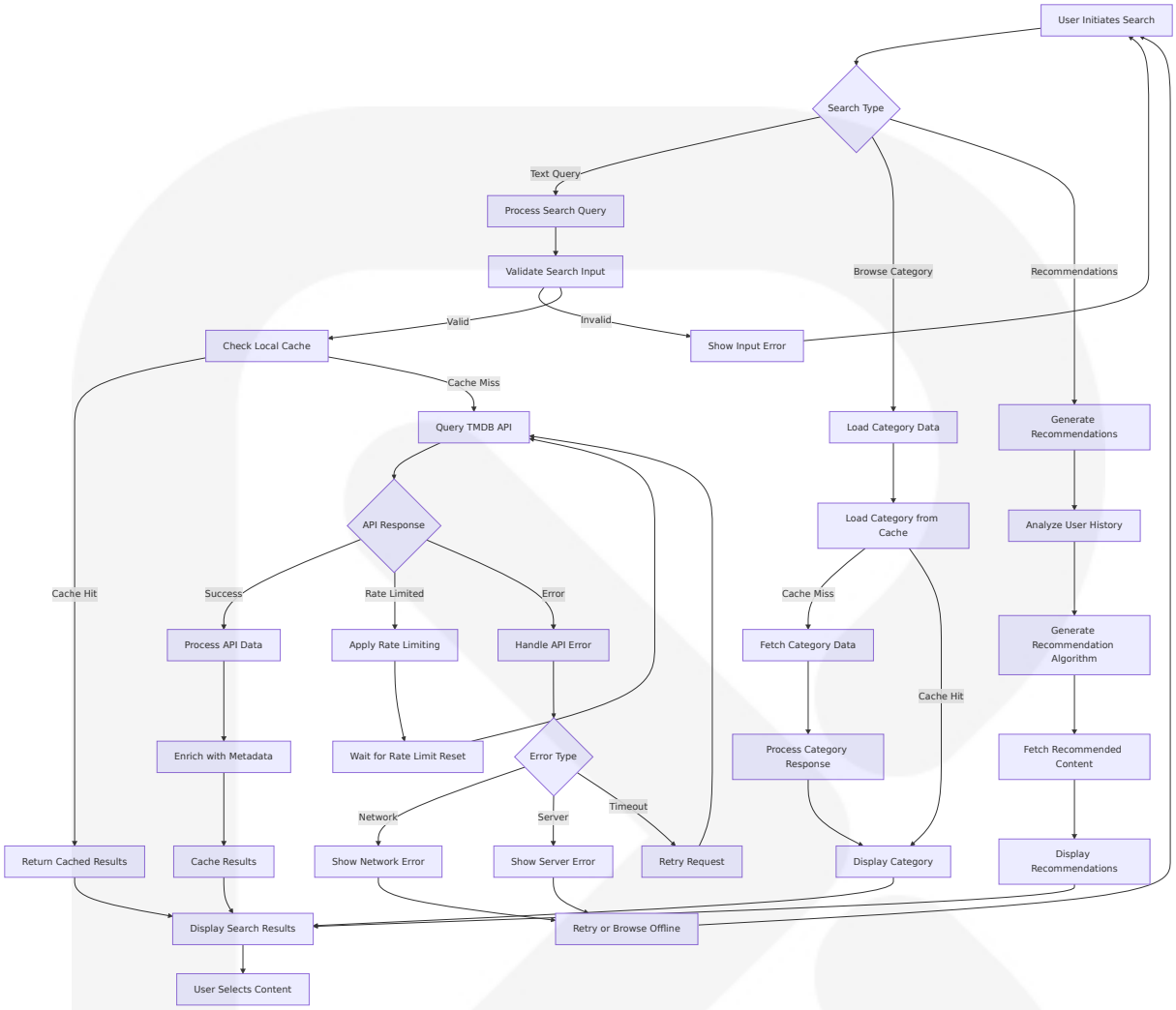
Business Rules:

- Guest mode requires no data whatsoever: in this mode, no calls are made to our backend. However, it comes at the expense of useful features, such as being able to sync your library across devices.
- Password complexity requirements: minimum 8 characters, mixed case, numbers
- Session timeout: 30 days for regular users, 24 hours for guest mode
- Maximum login attempts: 5 per IP address per hour

Error Handling:

- Network connectivity issues: Retry with exponential backoff
- Invalid credentials: Clear form and show specific error messages
- Server errors: Fallback to offline mode with limited functionality

Content Discovery and Search Flow



Performance Requirements:

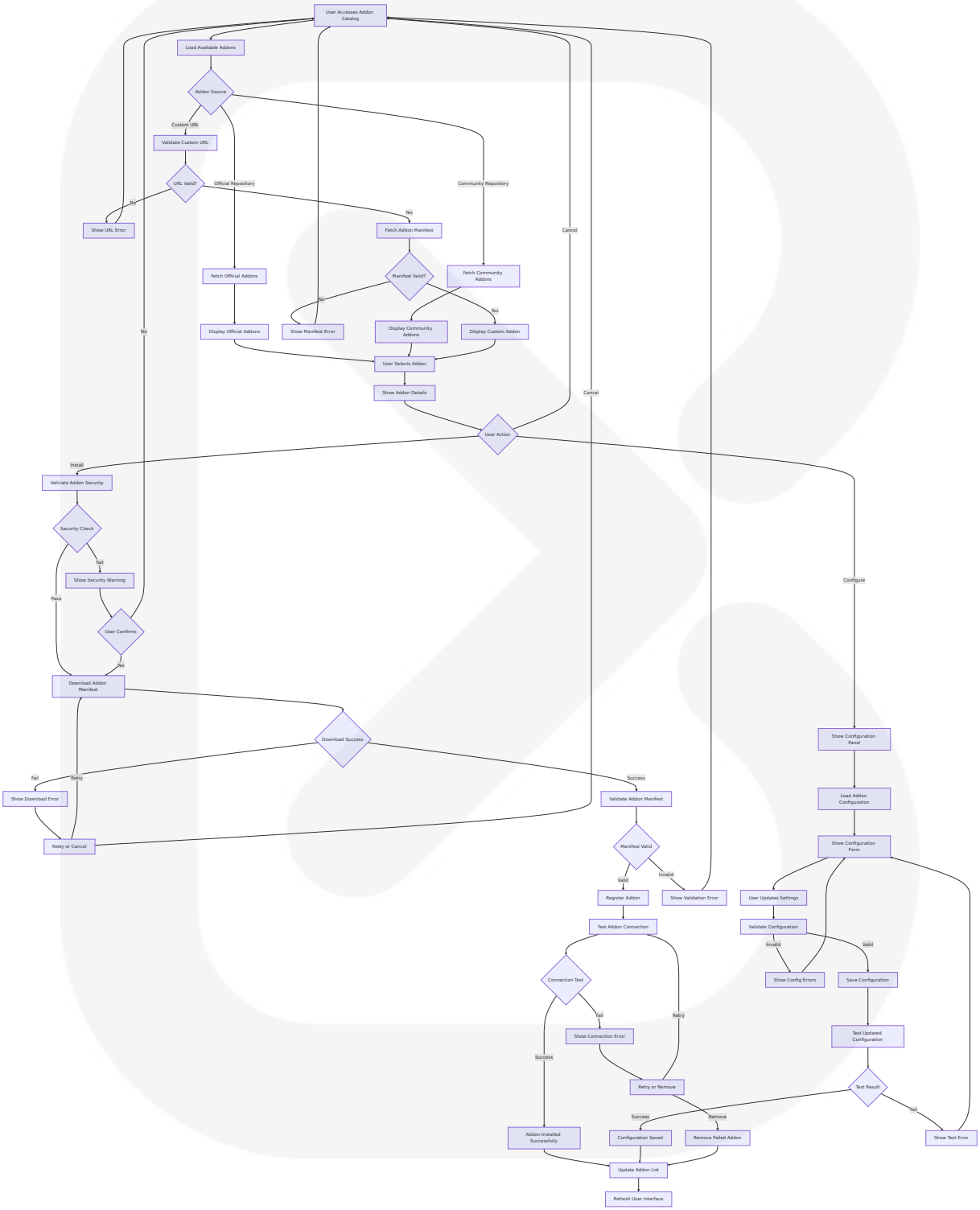
- TMDB API rate limits sit somewhere in the 50 requests per second range
- Maximum of 50 requests per second and 20 connections per IP
- Search response time: < 1 second for cached results, < 3 seconds for API calls
- Cache TTL: 24 hours for search results, 7 days for metadata

Data Validation:

- Minimum search query length: 2 characters
- Maximum search query length: 100 characters
- Input sanitization to prevent injection attacks

- API key validation before issuing requests

Addon Installation and Management Flow



Security Requirements:

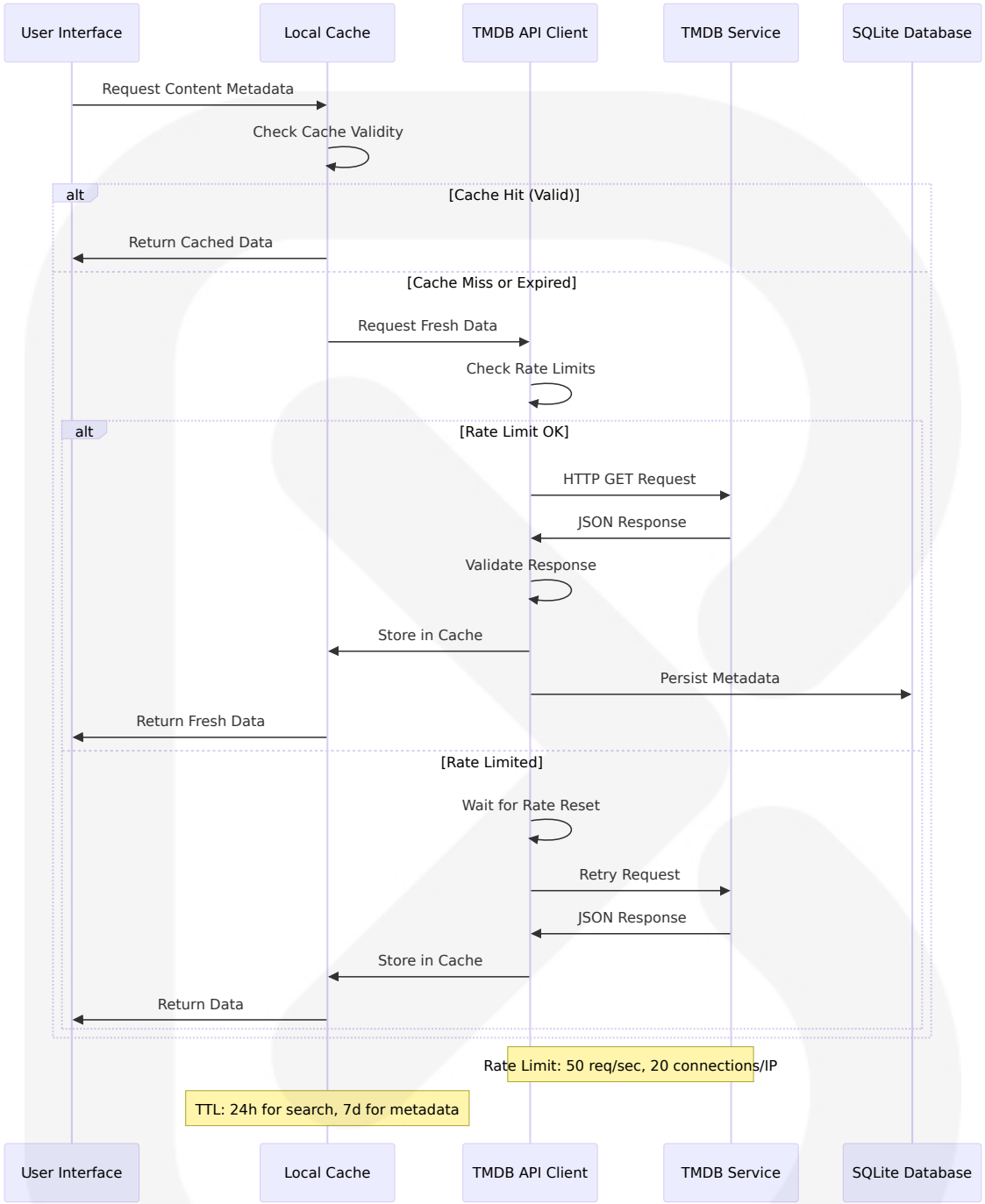
- Stremio's addon system was created with the user's security in mind. The addons do not run any code locally, so they pose no risks to your device.
- An add-on in Stremio doesn't generally run on a client's computer. Instead, it is hosted on the Internet just like any website. This brings ease of use and security benefits to the end user.
- If an add-on is served via HTTP, CORS headers must be present.
- HTTPS enforcement for all addon communications except localhost (127.0.0.1)
- Manifest schema validation to prevent malicious configurations
- URL sanitization and validation for custom addon sources

Validation Rules:

- The add-on must adhere to the add-on API. The most important part is the manifest. The add-on manifest is a JSON object describing the add-on's capabilities.
- Addon URLs must be accessible and return valid JSON manifests
- Configuration parameters must match expected data types and ranges
- Connection timeout: 10 seconds for addon availability tests

4.1.2 Integration Workflows

TMDB API Integration Flow



Rate Limiting Strategy:

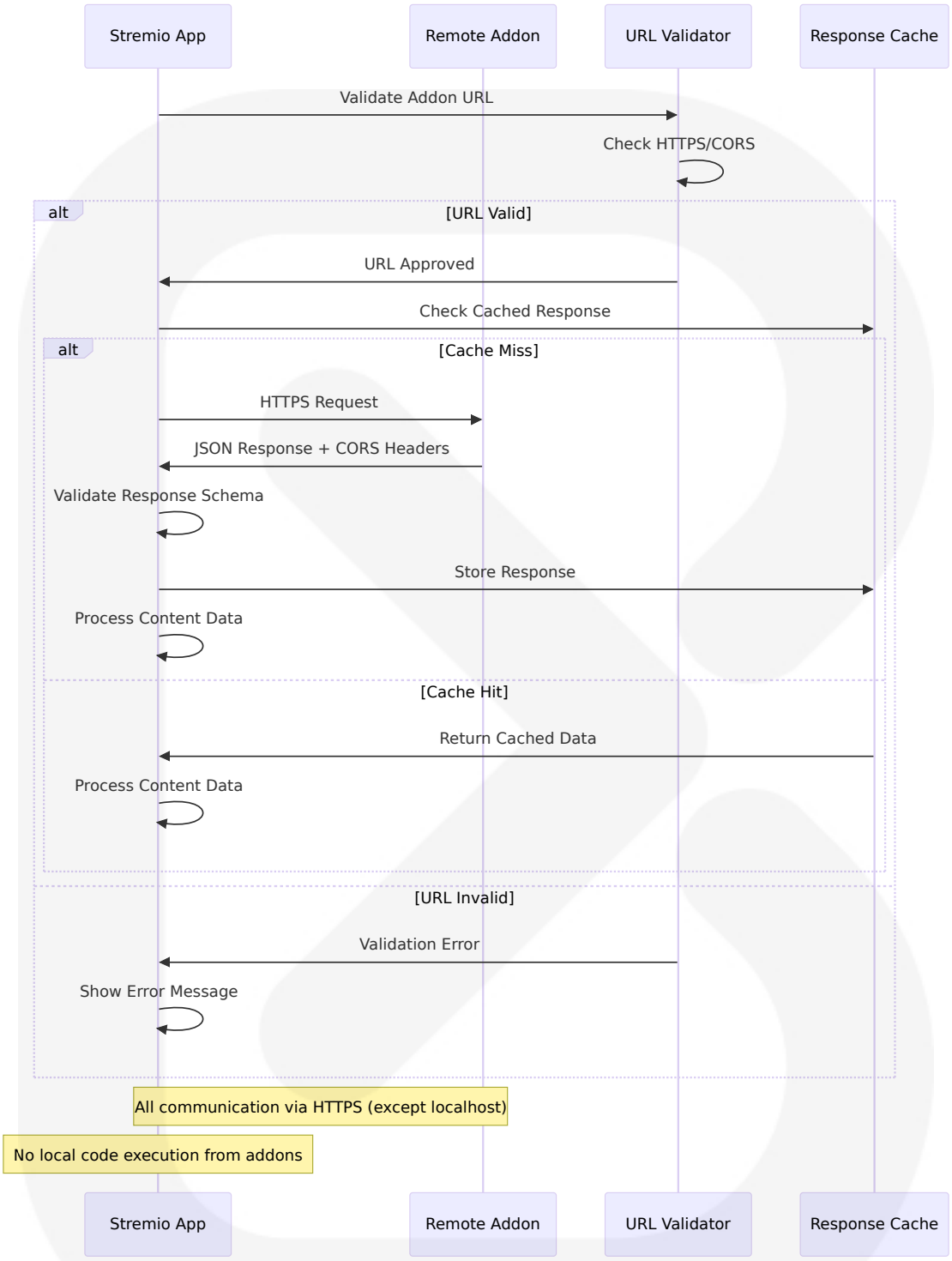
- TMDB still has some upper limits to help mitigate needlessly high bulk scraping. They sit somewhere in the 50 requests per second range.
- Maximum of 50 requests per second and 20 connections per IP
- Implement token bucket algorithm with 50 tokens per second
- Connection pooling with maximum 20 concurrent connections

- Exponential backoff for rate limit violations

Error Recovery:

- Network timeouts: 3 retry attempts with exponential backoff
- Server errors (5xx): Retry after 30 seconds
- Client errors (4xx): Log error and return cached data if available
- Legacy rate limits (40 requests every 10 seconds) have been disabled as of December 16, 2019

Addon Communication Flow



Security Validation:

- If an add-on is served via HTTP, CORS headers must be present
- HTTPS enforcement for all remote addons (localhost exceptions allowed)

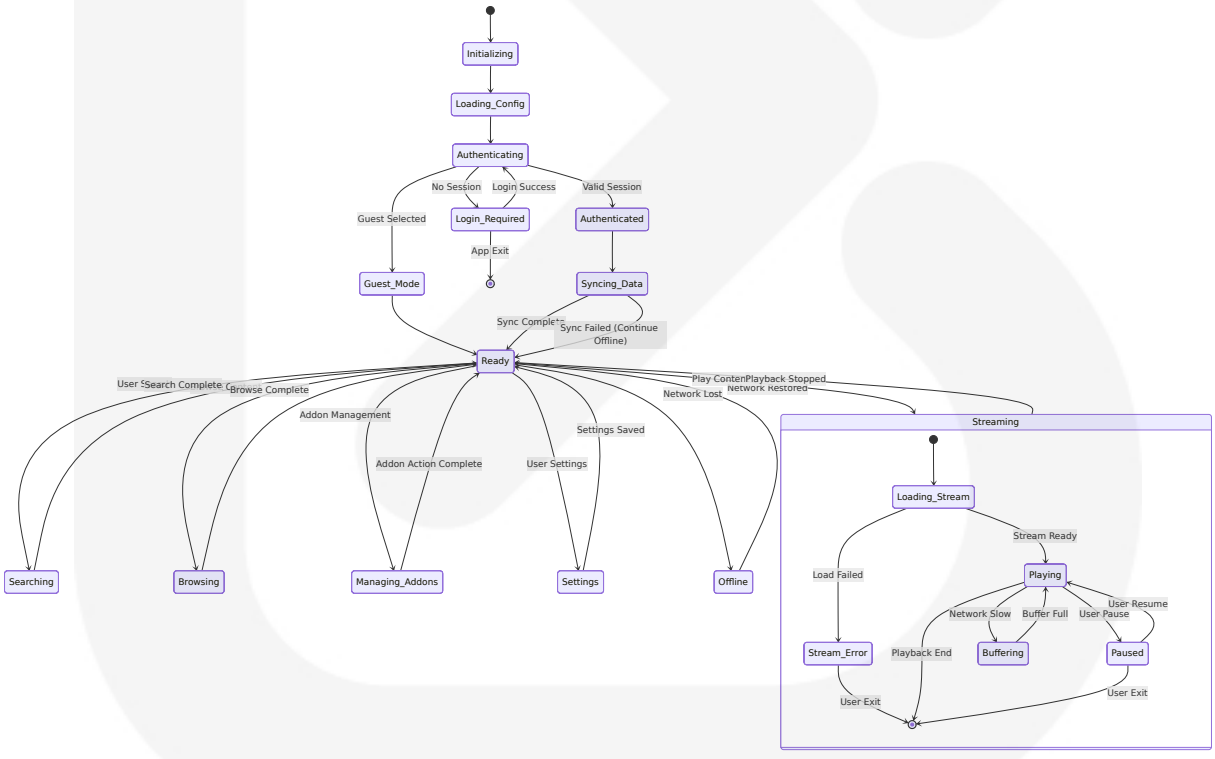
- Response schema validation against addon API specification
- Content-Type header validation (must be application/json)
- Response size limits to prevent DoS attacks

Performance Optimization:

- Response caching with 5-minute TTL for dynamic content
- Connection timeout: 10 seconds
- Read timeout: 30 seconds
- Maximum response size: 10MB

4.2 STATE MANAGEMENT

4.2.1 Application State Transitions



State Persistence:

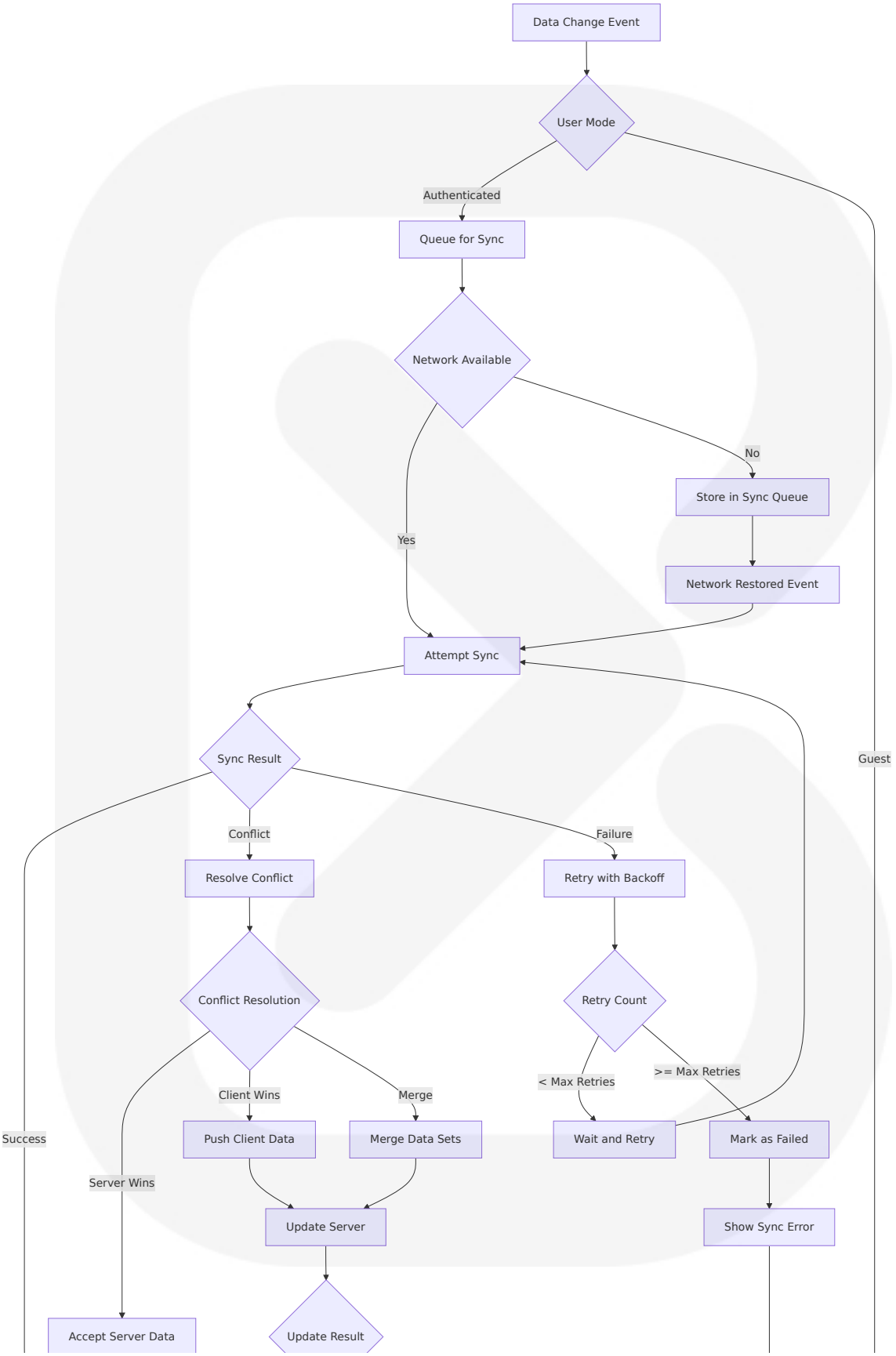
- User authentication state: Encrypted local storage
- Application preferences: SQLite database

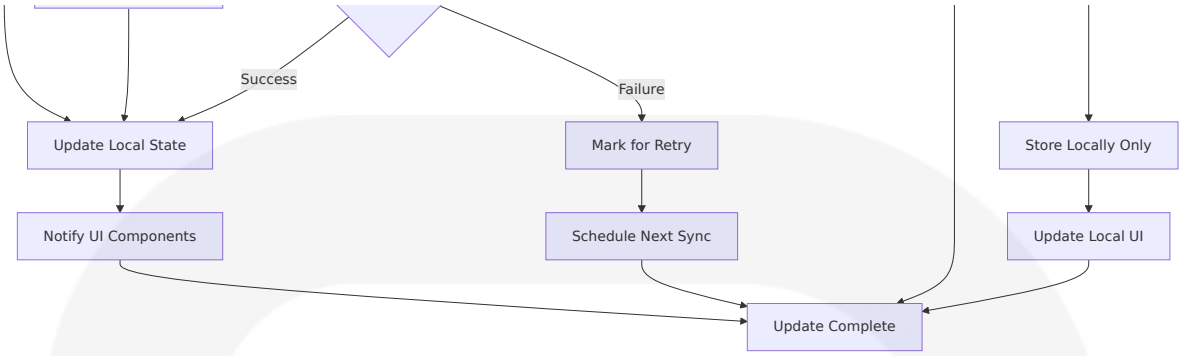
- Playback position: Real-time sync to local storage
- Addon configurations: Encrypted configuration files

State Recovery:

- Application crash: Restore last known good state
- Network interruption: Maintain offline functionality
- Invalid state: Reset to safe default state
- Corrupted data: Rebuild from backup or reset

4.2.2 Data Synchronization Flow


Data Change Event
User Mode
Authenticated
Guest
Queue for Sync
Network Available
No
Yes
Store in Sync Queue
Network Restored Event
Attempt Sync
Sync Result
Conflict
Failure
Success
Resolve Conflict
Retry with Backoff
Conflict Resolution
Retry Count
Client Wins
Merge
< Max Retries
>= Max Retries
Server Wins
Push Client Data
Merge Data Sets
Wait and Retry
Mark as Failed
Update Server
Show Sync Error
Accept Server Data
Update Result



Conflict Resolution Rules:

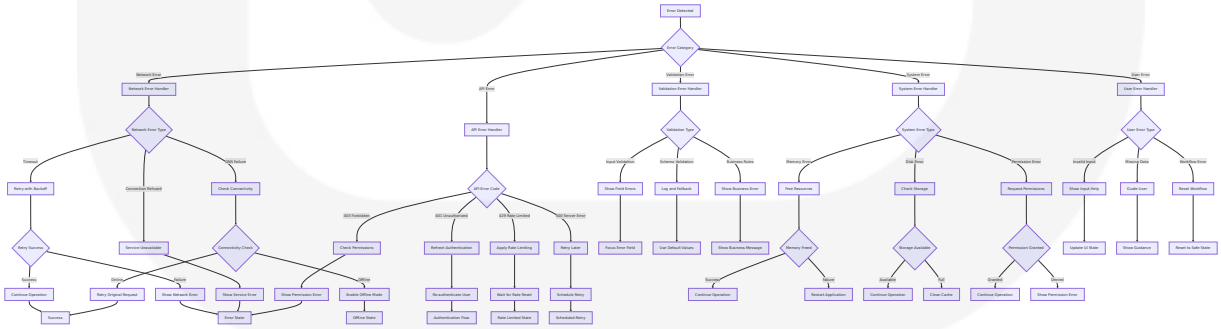
- Watch progress: Most recent timestamp wins
- Library additions: Merge both sets
- Settings changes: User preference with manual override option
- Addon configurations: Local changes take precedence

Sync Performance:

- Batch size: Maximum 100 items per sync operation
- Sync frequency: Every 5 minutes for active users
- Background sync: Every 30 minutes when app is idle
- Retry strategy: Exponential backoff with maximum 5 attempts

4.3 ERROR HANDLING AND RECOVERY

4.3.1 Error Classification and Response



Error Recovery Strategies:

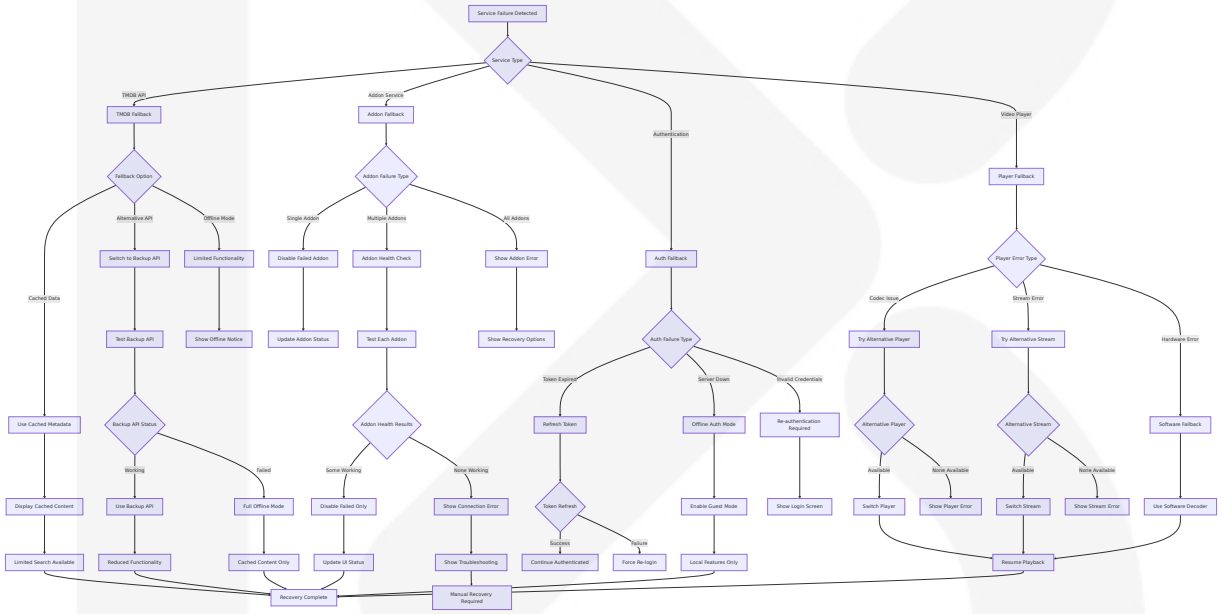
- Automatic retry with exponential backoff for transient errors

- Graceful degradation to offline mode when network unavailable
- User-friendly error messages with actionable guidance
- Comprehensive error logging for debugging and monitoring

Error Notification:

- Critical errors: Modal dialogs with clear actions
- Warning errors: Toast notifications with dismiss option
- Info errors: Status bar indicators
- Debug errors: Console logging only

4.3.2 Fallback and Recovery Procedures



Recovery Priorities:

1. **Critical Functions**: Authentication, basic navigation, cached content access
2. **Core Functions**: Search, content discovery, addon management
3. **Enhanced Functions**: Streaming, synchronization, advanced features
4. **Optional Functions**: Recommendations, social features, analytics

Recovery Timeouts:

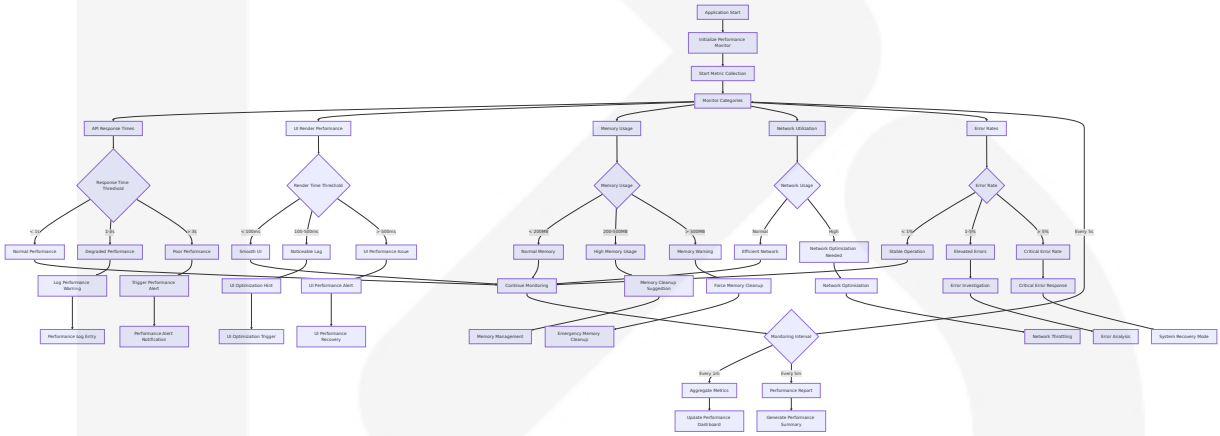
- Automatic recovery attempts: 3 retries over 5 minutes
- Service health checks: Every 30 seconds during outage
- Full system recovery: Maximum 10 minutes before manual intervention
- User notification: After 30 seconds of service unavailability

Data Integrity:

- Transaction rollback for failed operations
- Backup creation before critical operations
- Checksum validation for cached data
- Automatic corruption detection and repair

4.4 PERFORMANCE AND MONITORING

4.4.1 Performance Monitoring Flow



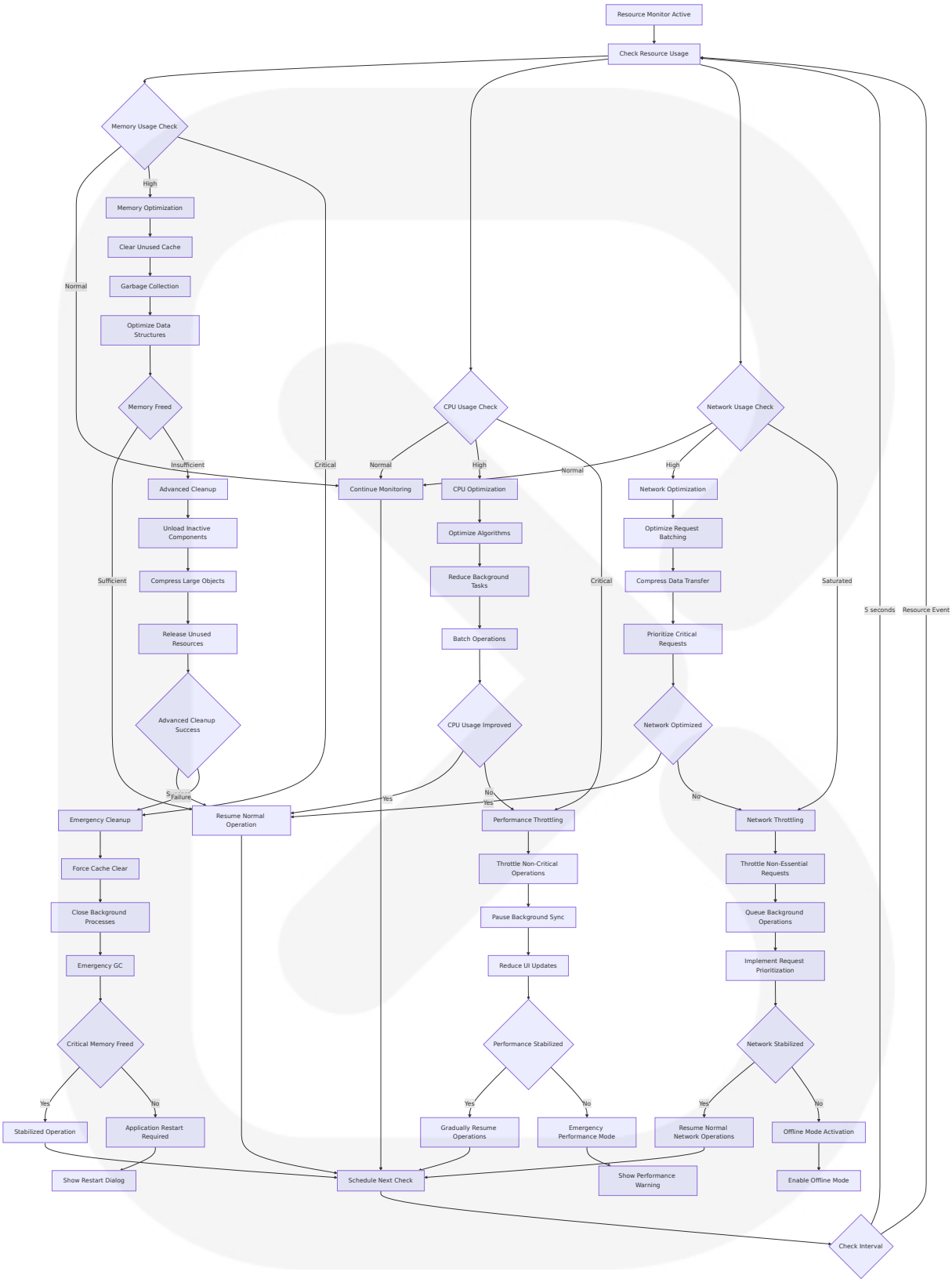
Performance Thresholds:

- API Response Time: < 1s excellent, 1-3s acceptable, > 3s poor
- UI Render Time: < 100ms smooth, 100-500ms noticeable, > 500ms problematic
- Memory Usage: < 200MB normal, 200-500MB high, > 500MB critical
- Error Rate: < 1% stable, 1-5% elevated, > 5% critical

Monitoring Intervals:

- Real-time metrics: Every 5 seconds
- Aggregated metrics: Every 1 minute
- Performance reports: Every 5 minutes
- Health checks: Every 30 seconds

4.4.2 Resource Management and Optimization



Resource Optimization Strategies:

- **Memory Management**: LRU cache eviction, object pooling, lazy loading
- **CPU Optimization**: Algorithm optimization, task batching, background processing
- **Network Efficiency**: Request batching, compression, connection pooling
- **Storage Management**: Cache rotation, temporary file cleanup, database optimization

Emergency Thresholds:

- Memory: > 80% of available system memory
- CPU: > 90% sustained usage for 30 seconds
- Network: > 95% bandwidth utilization
- Storage: < 100MB free space remaining

Recovery Actions:

- Automatic resource cleanup with user notification
- Graceful degradation of non-essential features
- Emergency mode with minimal functionality
- Application restart as last resort with user consent

5. SYSTEM ARCHITECTURE

5.1 HIGH-LEVEL ARCHITECTURE

5.1.1 System Overview

The Stremio Clone Desktop Application employs a modern hybrid architecture leveraging Tauri 2.0, which is a framework for building tiny and fast binaries for all major desktop (macOS, linux, windows) and mobile

(iOS, Android) platforms. Developers can integrate any frontend framework that compiles to HTML, JavaScript, and CSS for building their user experience while leveraging languages such as Rust, Swift, and Kotlin for backend logic when needed. In a Tauri application the frontend is written in your favorite web frontend stack. This runs inside the operating system WebView and communicates with the application core written mostly in Rust.

The architecture follows a **multi-process security model** with clear separation of concerns between the presentation layer (WebView) and the business logic layer (Rust Core). Tauri employs a multi-process architecture similar to Electron or many modern web browsers. This guide explores the reasons behind the design choice and why it is key to writing secure applications. It became clear that a more resilient architecture was needed, and applications began running different components in different processes. This makes much better use of modern multi-core CPUs and creates far safer applications. A crash in one component doesn't affect the whole system anymore, as components are isolated on different processes.

The system adopts a **remote addon execution model** inspired by Stremio's security-first approach. An add-on in Stremio, unlike other similar apps, doesn't generally run on a client's computer (however, there are exceptions). Instead, it is hosted on the Internet just like any website. This brings ease of use and security benefits to the end user. Stremio's addon system was also created with the user's security in mind. The addons do not run any code locally, so they pose no risks to your device.

Key Architectural Principles:

- **Security by Design**: Security best practices apply as well; for example, you must always sanitize user input, never handle secrets in the Frontend, and ideally defer as much business logic as possible to the Core process to keep your attack surface small.
- **Performance Optimization**: We hope to stabilize the core functionality and offer a stable framework, where the moving parts are

mostly plugins offering access to system specific functionality. You no longer need to understand all of Tauri to improve or implement specific features.

- **Cross-Platform Consistency**: Currently, Tauri uses Microsoft Edge WebView2 on Windows, WKWebView on macOS and webkitgtk on Linux.
- **Modular Plugin Architecture**: The plugins usually do not depend on other plugins, with some exceptions. This means to implement a new file system access functionality it is only required to contribute to the fs plugin instead of Tauri itself.

5.1.2 Core Components Table

Component Name	Primary Responsibility	Key Dependencies	Integration Points	Critical Considerations
Tauri Core Process	System integration, IPC coordination, security enforcement	Rust runtime, SQLite, HTTP client	WebView IPC, System APIs, Plugin system	Process isolation, memory safety, secure IPC
WebView Frontend	User interface rendering, user interaction handling	System WebView, Frontend framework	Tauri IPC bridge, DOM APIs	Cross-platform compatibility, security boundaries
Addon Communication Layer	Remote addon discovery, manifest validation, stream processing	HTTPS client, CORS validation	Remote addon servers, Content validators	Security validation, network resilience
Content Discovery Engine	Metadata aggregation, search functionality, caching	TMDB API client, SQLite cache	External APIs, Local database	Rate limiting, cache invalidation, offline support

5.1.3 Data Flow Description

The primary data flow follows a **secure message-passing pattern** between isolated processes. Tauri uses a particular style of Inter-Process Communication called Asynchronous Message Passing, where processes exchange requests and responses serialized using some simple data representation. The primary API, invoke, is similar to the browser's fetch API and allows the Frontend to invoke Rust functions, pass arguments, and receive data. Because this mechanism uses a JSON-RPC like protocol under the hood to serialize requests and responses, all arguments and return data must be serializable to JSON.

Content Discovery Flow: User search requests originate in the WebView, pass through the secure IPC layer to the Rust Core, which queries the TMDB API and local cache. Results are processed, cached using SQLite's JSONB format for performance, and returned to the frontend for display.

Addon Integration Flow: The add-on must adhere to the add-on API. The most important part is the manifest. The add-on manifest is a JSON object describing the add-on's capabilities. Every resource is accessed at a certain endpoint where your add-on should respond with proper data. Addon manifests are fetched over HTTPS, validated against schema, and registered in the local database. Content streams are requested from remote addons and validated before being passed to the video player.

Data Persistence Strategy: Beginning with version 3.45.0 (2024-01-15), SQLite allows its internal "parse tree" representation of JSON to be stored on disk, as a BLOB, in a format that we call "JSONB". By storing SQLite's internal binary representation of JSON directly in the database, applications can bypass the overhead of parsing and rendering JSON when reading and updating JSON values. The internal JSONB format also uses slightly less disk space then text JSON.

5.1.4 External Integration Points

System Name	Integration Type	Data Exchange Pattern	Protocol/Format	SLA Requirements
TMDB API	RESTful API	Request/Response with caching	HTTPS/JSON	50 req/sec, no SLA provided
Remote Addons	HTTP API	Manifest + Stream URLs	HTTPS/JSON with CORS	10 second timeout, HTTPS required
System WebView	Native Integration	IPC Message Passing	JSON-RPC over secure channel	< 100ms response time
Local SQLite	Embedded Database	Direct SQL queries	SQLite with JSONB	< 1ms query response

5.2 COMPONENT DETAILS

5.2.1 Tauri Core Process

Purpose and Responsibilities:
The Tauri Core serves as the secure backend orchestrator, managing all system interactions, data persistence, and business logic execution. The Core's primary responsibility is to use that access to create and orchestrate application windows, system-tray menus, or notifications. Tauri implements the necessary cross-platform abstractions to make this easy. It also routes all Inter-Process Communication through the Core process, allowing you to intercept, filter, and manipulate IPC messages in one central place. The Core process should also be responsible for managing global state, such as settings or database connections. This allows you to easily synchronize state between windows and protect your business-sensitive data from prying eyes in the Frontend.

Technologies and Frameworks:

- **Rust 1.70+ (MSRV)**: We chose Rust to implement Tauri because of its concept of Ownership guarantees memory safety while retaining excellent performance.
- **Tokio Runtime**: Asynchronous I/O and task scheduling
- **SQLite with JSONB**: The advantage of JSONB in SQLite is that it is smaller and faster than text JSON - potentially several times faster.
- **reqwest HTTP Client**: External API communication with connection pooling

Key Interfaces and APIs:

- **Tauri Commands**: Type-safe function calls exposed to frontend
- **Event System**: Bidirectional event communication
- **Plugin System**: Modular functionality extensions
- **Database Layer**: SQLite operations with JSONB optimization

Data Persistence Requirements:

- User authentication tokens (encrypted)
- Content metadata cache with TTL
- Addon configurations and manifests
- Application settings and preferences

Scaling Considerations:

- Connection pooling for HTTP clients
- Database connection management
- Memory-efficient caching strategies
- Background task scheduling

5.2.2 WebView Frontend Layer

Purpose and Responsibilities:
The WebView Frontend provides the user interface and handles all user interactions while maintaining security boundaries. It operates within the

system's native WebView component and communicates exclusively through Tauri's secure IPC system.

Technologies and Frameworks:

- **System WebView**: Platform-native rendering engine
- **Modern Web Standards**: HTML5, CSS3, ES2022+
- **Frontend Framework**: Framework-agnostic (React/Vue/Svelte supported)
- **Tauri JavaScript API**: Secure IPC communication layer

Key Interfaces and APIs:

- **Tauri Invoke API**: Command execution interface
- **Event Listeners**: Real-time updates from backend
- **WebView APIs**: DOM manipulation and user interaction
- **Media APIs**: Video playback and casting integration

Security Considerations:

- Content Security Policy enforcement
- Input sanitization before IPC calls
- No direct system access capabilities
- Isolated execution environment

5.2.3 Addon Communication System

Purpose and Responsibilities:
Manages the discovery, validation, and communication with remote Stremio-compatible addons while ensuring security and performance standards.

Technologies and Frameworks:

- **HTTPS Client**: Secure remote communication
- **JSON Schema Validation**: Manifest verification

- **CORS Handling**: Cross-origin request management
- **URL Validation**: Security-focused endpoint verification

Key Interfaces and APIs:

- **Addon Discovery**: Repository browsing and search
- **Manifest Validation**: Schema compliance checking
- **Stream Resolution**: Content URL retrieval
- **Health Monitoring**: Addon availability tracking

Security Requirements:
Please note: addon URLs in Stremio must be loaded with HTTPS (except 127.0.0.1) and must support CORS! CORS support is handled automatically by the SDK, but if you're trying to load your addon remotely (not from 127.0.0.1), you need to support HTTPS.

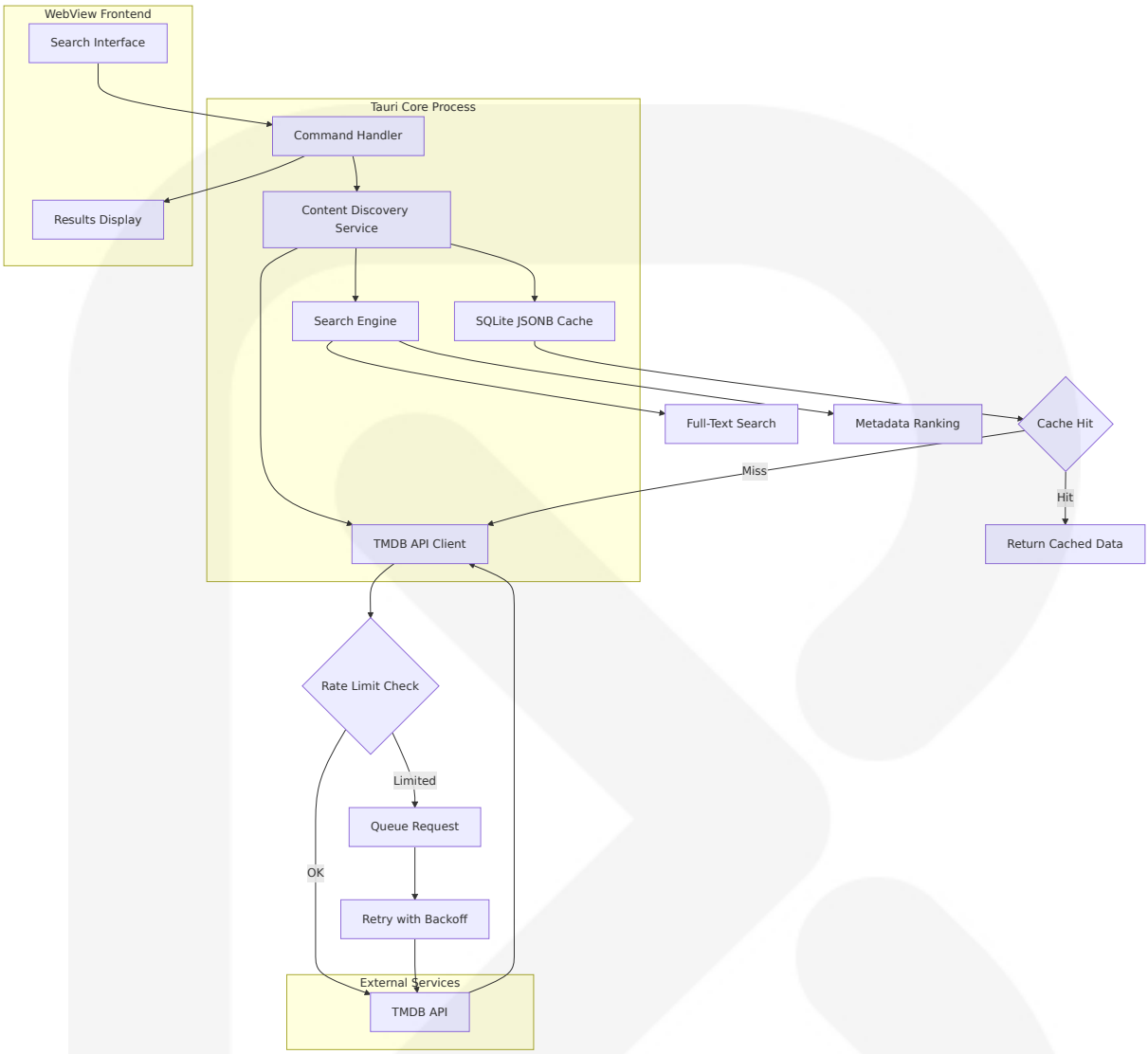
5.2.4 Content Discovery Engine

Purpose and Responsibilities:
Aggregates content metadata from multiple sources, provides search functionality, and maintains local caching for optimal performance.

Technologies and Frameworks:

- **TMDB API Integration**: Primary metadata source
- **SQLite JSONB Storage**: High-performance caching
- **Search Algorithms**: Full-text search with ranking
- **Cache Management**: TTL-based invalidation

Performance Optimizations:
If the input is already in the JSONB format, no translation is needed, that step can be skipped, and performance is faster. For that reason, when an argument to one JSON function is supplied by another JSON function, it is usually more efficient to use the "jsonb_" variant for the function used as the argument.



5.3 TECHNICAL DECISIONS

5.3.1 Architecture Style Decisions

Multi-Process Architecture Selection:

Decision Factor	Rationale	Trade-offs
Security Isolation	Message passing is a safer technique than shared memory or direct function access because the recipient is free to reject or	Slight performance overhead vs. en

Decision Factor	Rationale	Trade-offs
	discard requests as it sees fit. For example, if the Tauri Core process determines a request to be malicious, it simply discards the requests and never executes the corresponding function.	hanced security
Process Resilience	A crash in one component doesn't affect the whole system anymore, as components are isolated on different processes. If a process gets into an invalid state, we can easily restart it. We can also limit the blast radius of potential exploits by handing out only the minimum amount of permissions to each process, just enough so they can get their job done.	Memory overhead vs. fault tolerance
Cross-Platform Consistency	Unified architecture across Windows, macOS, and Linux	Development complexity vs. maintenance efficiency

Remote Addon Execution Model:

The decision to adopt Stremio's remote addon architecture provides significant security advantages over traditional plugin systems that execute code locally. This architectural choice eliminates entire classes of security vulnerabilities while maintaining extensibility.

5.3.2 Communication Pattern Choices

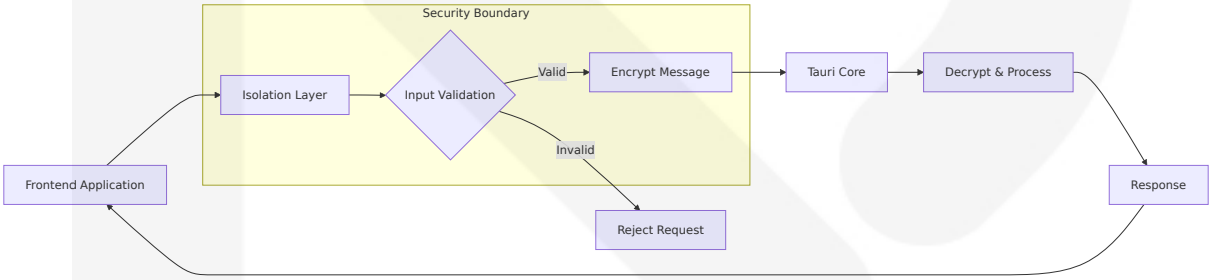
IPC Pattern Selection:

Tauri uses a particular style of Inter-Process Communication called Asynchronous Message Passing, where processes exchange requests and responses serialized using some simple data representation. Message Passing should sound familiar to anyone with web development

experience, as this paradigm is used for client-server communication on the internet.

Security Enhancement with Isolation Pattern:

Tauri highly recommends using the isolation pattern whenever it can be used. Because the Isolation application intercepts all messages from the frontend, it can always be used. Tauri also strongly suggests locking down your application whenever you use external Tauri APIs. As the developer, you can utilize the secure Isolation application to try and verify IPC inputs, to make sure they are within some expected parameters.



5.3.3 Data Storage Solution Rationale

SQLite with JSONB Selection:

Requirement	SQLite Advantage	JSONB Enhancement
Performance	The advantage of JSONB over ordinary text RFC 8259 JSON is that JSONB is both slightly smaller (by between 5% and 10% in most cases) and can be processed in less than half the number of CPU cycles.	Significant CPU reduction for JSON operations
Storage Efficiency	Embedded database with no server overhead	5-10% space savings over text JSON
Cross-Platform	Single file database, portable across platforms	Consistent performance characteristics

Requirement	SQLite Advantage	JSONB Enhancement
Caching Strategy	Built-in transaction support with WAL mode	Optimized for frequent read/write patterns

Database Schema Design:

The schema leverages SQLite's JSONB capabilities for flexible metadata storage while maintaining relational integrity for core entities like users, addons, and content references.

5.3.4 Caching Strategy Justification

Multi-Layer Caching Architecture:

Cache Layer	Purpose	TTL Strategy	Invalidation Method
HTTP Response Cache	API call optimization	5 minutes for dynamic content	Time-based expiration
Metadata Cache	Content information storage	24 hours for search, 7 days for details	Manual refresh + TTL
Image Cache	Poster/backdrop storage	30 days	LRU eviction + size limits
Addon Manifest Cache	Plugin configuration	1 hour	Version-based invalidation

Performance Optimization Strategy:

Most JSON functions do their internal processing using JSONB. So if the input is text, they first must translate the input text into JSONB. If the input is already in the JSONB format, no translation is needed, that step can be skipped, and performance is faster.

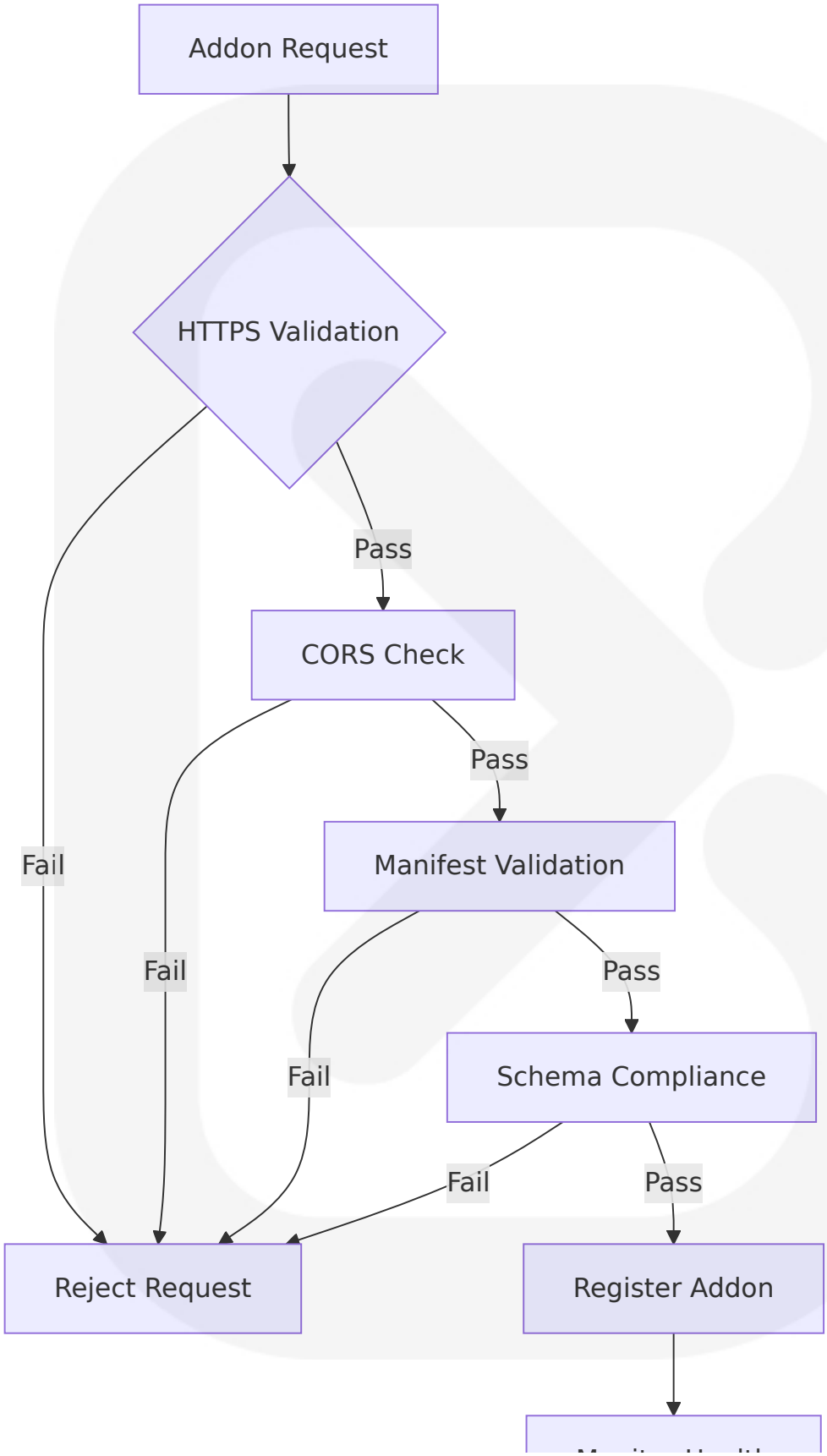
5.3.5 Security Mechanism Selection

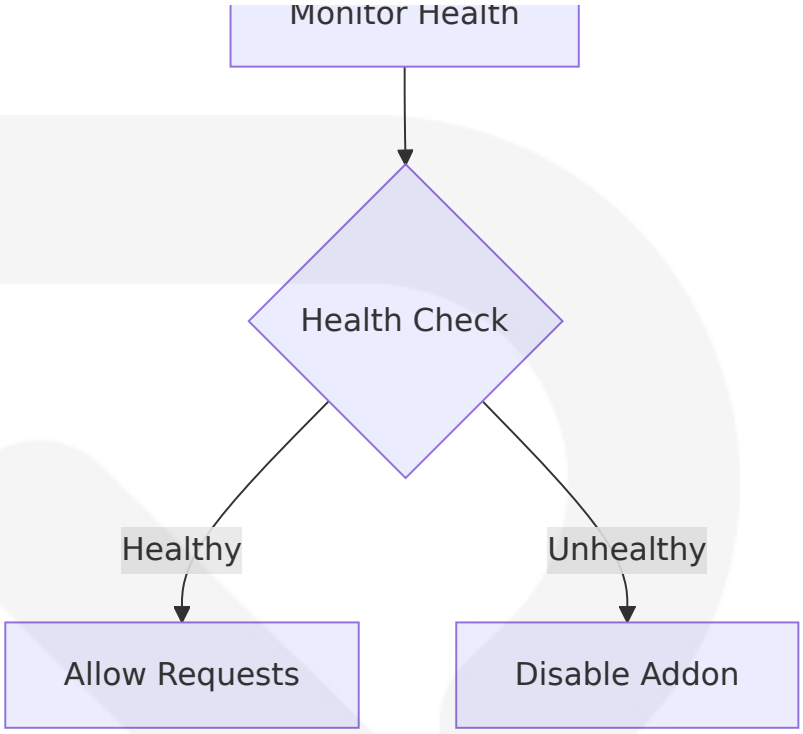
Trust Boundary Implementation:

Tauri's security model differentiates between Rust code written for the application's core and frontend code written in any framework or language understood by the system WebView. Inspecting and strongly defining all data passed between boundaries is very important to prevent trust boundary violations. If data is passed without access control between these boundaries then it's easy for attackers to elevate and abuse privileges. The IPC layer is the bridge for communication between these two trust groups and ensures that boundaries are not broken.

Addon Security Model:

The remote execution model eliminates local code execution risks while maintaining functionality through secure HTTP-based communication with proper validation and sandboxing.


Addon Request
HTTPS Validation
Pass
CORS Check
Pass
Manifest Validation
Pass
Schema Compliance
Pass
Register Addon
Fail
Fail
Fail
Fail
Fail
Reject Request



5.4 CROSS-CUTTING CONCERNS

5.4.1 Monitoring and Observability Approach

Performance Monitoring Strategy:

The system implements comprehensive monitoring across all architectural layers, focusing on user experience metrics, system resource utilization, and external service health.

Key Metrics Collection:

Metric Category	Collection Method	Alerting Thresholds	Storage Strategy
IPC Performance	Built-in Tauri metrics	> 100ms response time	Local SQLite logging
API Response Times	HTTP client instrumentation	> 3 seconds for TMDB	In-memory aggregation

Metric Category	Collection Method	Alerting Thresholds	Storage Strategy
Cache Hit Rates	Database query analysis	< 80% hit rate	Performance counters
Memory Usage	System resource monitoring	> 500MB sustained	Real-time tracking

Observability Implementation:

The monitoring system leverages Rust's built-in performance profiling capabilities combined with custom metrics collection for application-specific concerns like addon health and content discovery performance.

5.4.2 Logging and Tracing Strategy

Structured Logging Architecture:

- **Frontend Logging**: Console-based logging with configurable levels
- **Backend Logging**: Structured logging with `tracing` crate integration
- **Audit Logging**: Security-relevant events with tamper-evident storage
- **Performance Tracing**: Request/response timing with correlation IDs

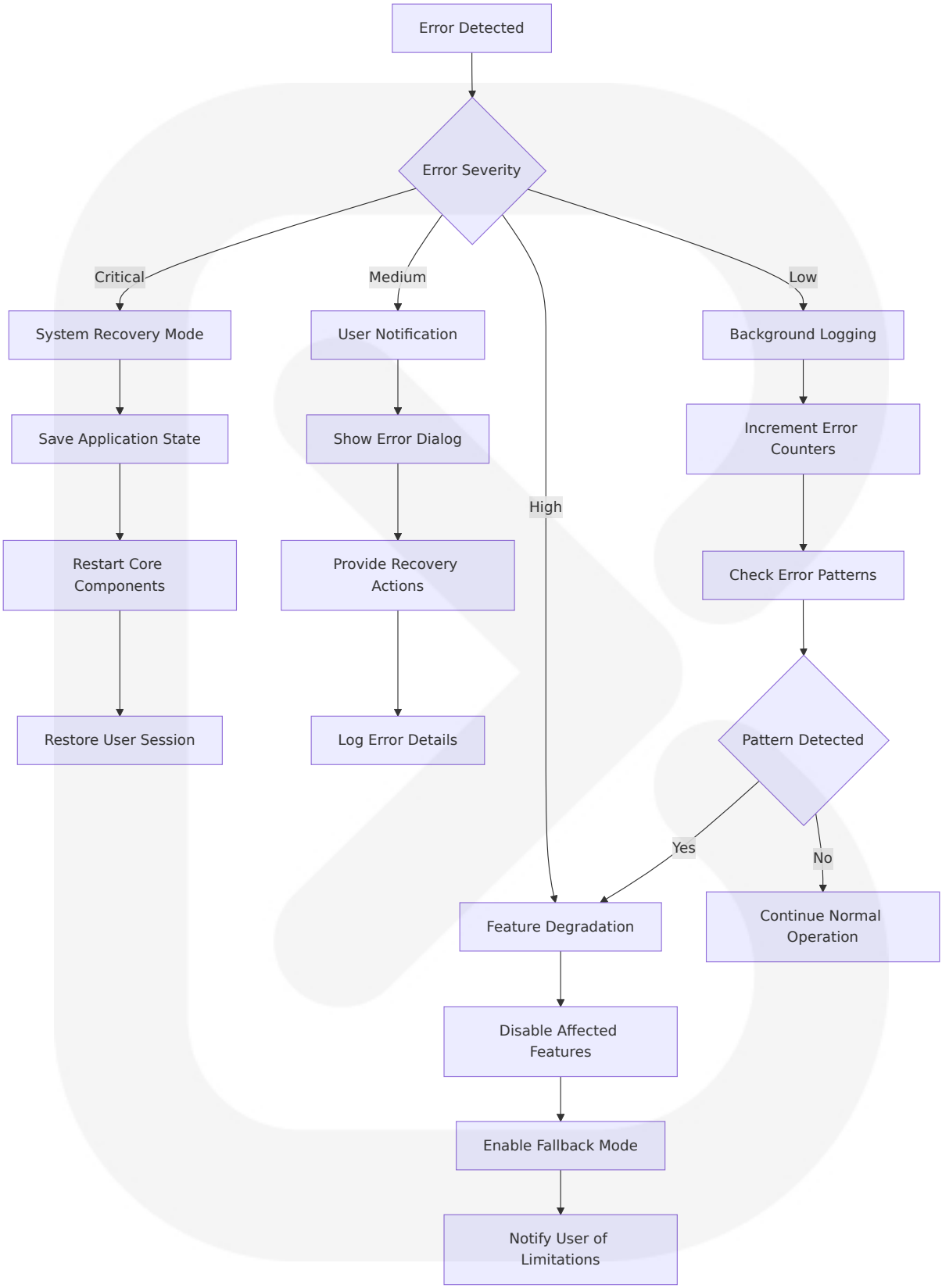
Log Retention and Management:

Log Type	Retention Period	Storage Location	Privacy Considerations
Application Logs	30 days	Local file system	No PII logging
Security Events	90 days	Encrypted local storage	Audit trail integrity
Performance Metrics	7 days	In-memory + periodic export	Aggregated data only
Debug Traces	24 hours	Memory buffer	Development builds only

5.4.3 Error Handling Patterns

Hierarchical Error Management:

The system implements a comprehensive error handling strategy that provides graceful degradation while maintaining security and user experience standards.



Error Recovery Mechanisms:

- **Network Failures**: Automatic retry with exponential backoff
- **API Rate Limiting**: Request queuing with priority scheduling
- **Cache Corruption**: Automatic cache rebuild with user notification
- **Addon Failures**: Individual addon isolation with health monitoring

5.4.4 Authentication and Authorization Framework

Multi-Tier Authentication System:

Authentication Level	Implementation	Security Features	Fallback Strategy
Local Authentication	SQLite-based credential storage	Bcrypt hashing, session tokens	Guest mode activation
Cloud Synchronization	Optional remote authentication	OAuth 2.0, refresh tokens	Local-only operation
Addon Authorization	URL-based validation	HTTPS enforcement, CORS validation	Addon disable/removal
Content Access	Stream URL validation	Input sanitization, timeout limits	Error reporting

Authorization Patterns:

The system implements capability-based security where each component has minimal required permissions. Frontend components cannot directly access system resources, and all privileged operations are mediated through the secure IPC layer.

5.4.5 Performance Requirements and SLAs

System Performance Targets:

Performance Metric	Target Value	Measurement Method	Degradation Response
Application Startup	< 3 seconds	Automated timing	Optimize initialization sequence
Search Response	< 1 second	User interaction tracking	Enable progressive loading
Video Stream Start	< 5 seconds	Media player integration	Implement stream preloading
IPC Communication	< 100ms	Built-in Tauri metrics	Optimize message serialization

Resource Utilization Limits:

- **Memory Usage**: < 200MB normal operation, < 500MB peak
- **CPU Usage**: < 10% idle, < 50% during intensive operations
- **Network Bandwidth**: Adaptive based on connection quality
- **Storage Growth**: < 1GB cache size with automatic cleanup

5.4.6 Disaster Recovery Procedures

Data Protection Strategy:

Data Category	Backup Method	Recovery Time	Recovery Point
User Profiles	Local + optional cloud sync	< 1 minute	Last sync point
Content Cache	Rebuildable from sources	< 5 minutes	Full rebuild
Application Settings	Local file backup	< 30 seconds	Last application run
Addon Configurations	Export/import functionality	< 2 minutes	Last configuration save

Recovery Procedures:

The system implements automatic recovery mechanisms for common failure scenarios, with manual recovery options for complex situations. All recovery procedures maintain data integrity while minimizing user disruption.

Business Continuity Measures:

- **Offline Mode**: Core functionality available without network connectivity
- **Graceful Degradation**: Non-essential features disabled during resource constraints
- **State Preservation**: Application state maintained across crashes and restarts
- **User Data Protection**: Encrypted local storage with corruption detection

6. SYSTEM COMPONENTS DESIGN

6.1 CORE ARCHITECTURE COMPONENTS

6.1.1 Tauri Application Framework

Component Overview:
Tauri is a framework for building tiny and fast binaries for all major desktop (macOS, linux, windows) and mobile (iOS, Android) platforms. Developers can integrate any frontend framework that compiles to HTML, JavaScript, and CSS for building their user experience while leveraging languages such as Rust, Swift, and Kotlin for backend logic when needed. In a Tauri application the frontend is written in your favorite web frontend stack. This

runs inside the operating system WebView and communicates with the application core written mostly in Rust.

Technical Implementation:

Component Layer	Technology Stack	Responsibility	Performance Characteristics
Frontend Process	System WebView + Web Technologies	UI rendering, user interaction	< 100ms response time for interactions
Core Process	Rust + Tokio Runtime	Business logic, system integration	< 10ms for IPC command processing
IPC Bridge	JSON-RPC over secure channels	Inter-process communication	Asynchronous message passing
Plugin System	Tauri Plugin Architecture	System API access	Modular, isolated functionality

Security Architecture:
The multi-process architecture provides enhanced security through process isolation. We hope to stabilize the core functionality and offer a stable framework, where the moving parts are mostly plugins offering access to system specific functionality. You no longer need to understand all of Tauri to improve or implement specific features. The plugins usually do not depend on other plugins, with some exceptions. This means to implement a new file system access functionality it is only required to contribute to the fs plugin instead of Tauri itself.

Cross-Platform Compatibility:
The framework leverages native WebView components for optimal performance and consistency across platforms. Currently, Tauri uses Microsoft Edge WebView2 on Windows, WKWebView on macOS and webkitgtk on Linux, ensuring native performance while maintaining a unified development experience.

6.1.2 Content Discovery and Metadata Engine

TMDB API Integration Component:

While our legacy rate limits have been disabled for some time, we do still have some upper limits to help mitigate needlessly high bulk scraping. They sit somewhere in the 50 requests per second range. The content discovery engine implements intelligent rate limiting and caching strategies to optimize API usage.

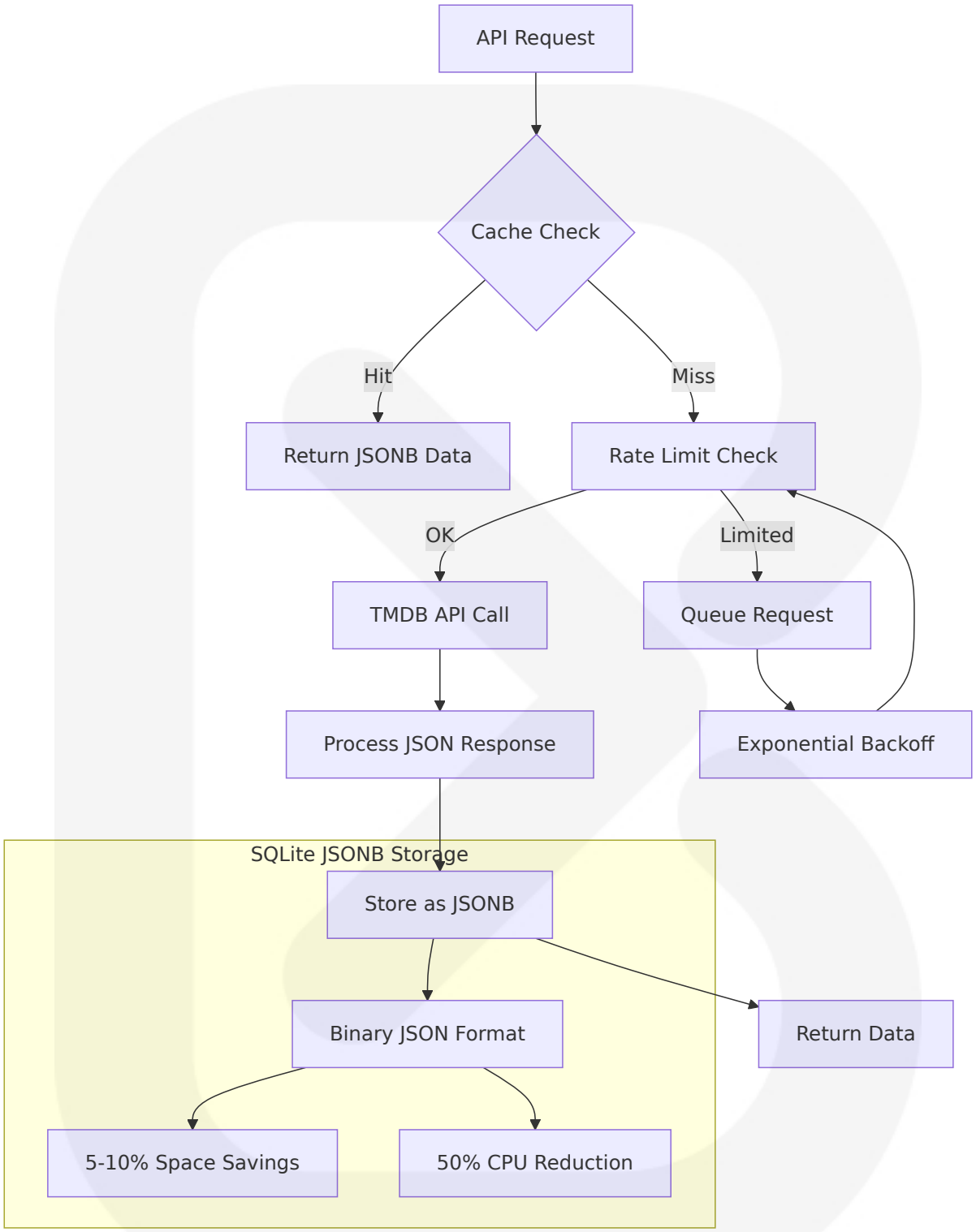
Rate Limiting Implementation:

Rate Limit Type	Specification	Implementation Strategy
API Requests	I believe it's a maximum of: 50 requests per second and 20 connections per IP.	Token bucket algorithm with connection pooling
Image Requests	For image.tmdb.org the only thing we limit is the max number of simultaneous connections. The limit is the same, 20.	Connection reuse with keep-alive
Legacy Limits	As of December 16, 2019, we have disabled the original API rate limiting (40 requests every 10 seconds.)	No longer applicable

Caching Strategy with SQLite JSONB:

The advantage of JSONB over ordinary text RFC 8259 JSON is that JSONB is both slightly smaller (by between 5% and 10% in most cases) and can be processed in less than half the number of CPU cycles. The metadata cache leverages SQLite's JSONB format for optimal performance.

Performance Optimization:



Attribution Requirements:

Our API is free to use for non-commercial purposes as long as you attribute TMDB as the source of the data and/or images. You shall place the

following notice prominently on your application: "This product uses the TMDB API but is not endorsed or certified by TMDB."

6.1.3 Addon System Architecture

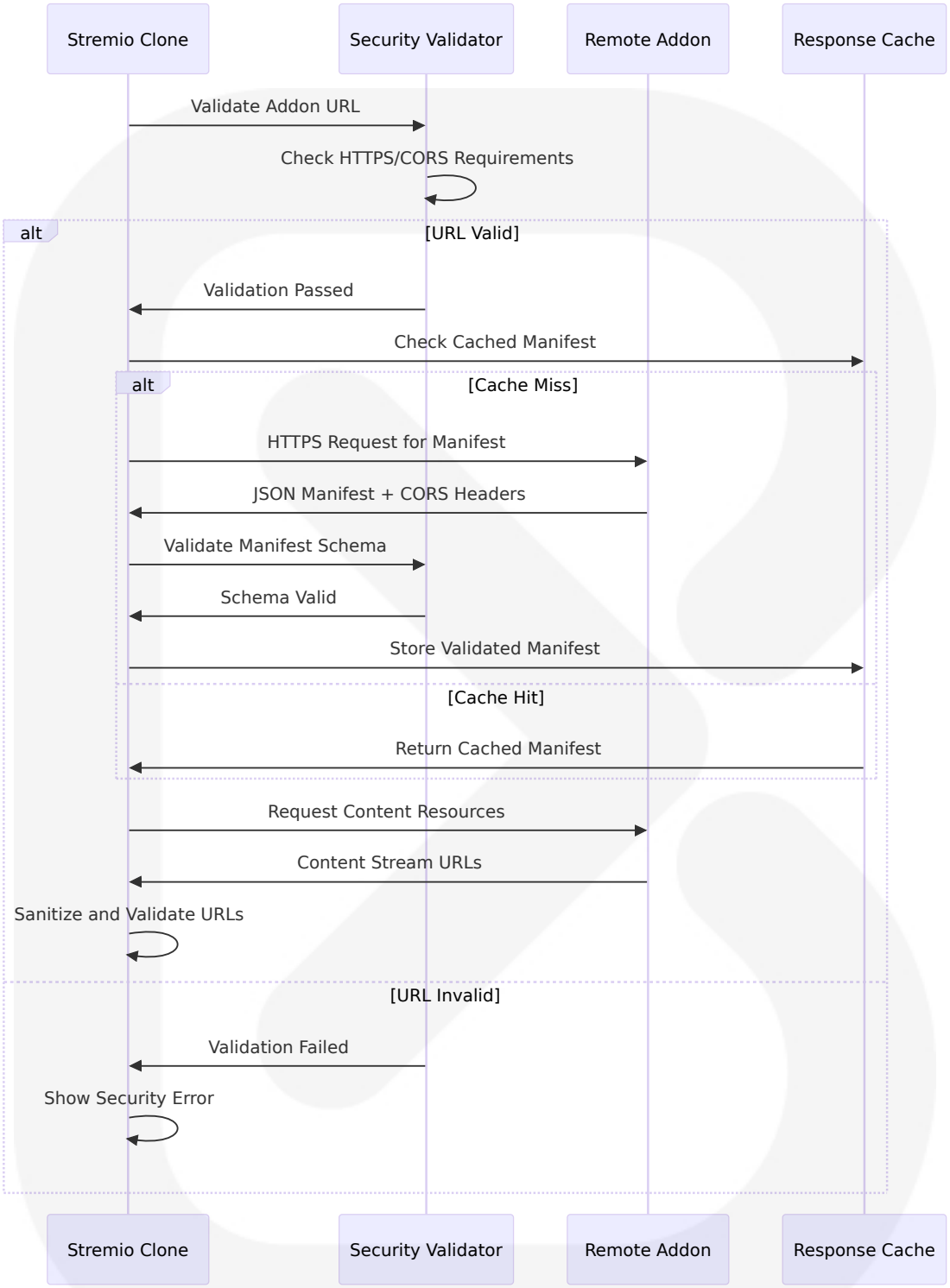
Remote Execution Security Model:

An add-on in Stremio, unlike other similar apps, doesn't generally run on a client's computer (however, there are exceptions). Instead, it is hosted on the Internet just like any website. This brings ease of use and security benefits to the end user.

Security Implementation:

Security Layer	Requirement	Validation Method
HTTPS Enforcement	If an add-on is served via HTTP, CORS headers must be present.	URL protocol validation
Manifest Validation	The add-on must adhere to the add-on API. The most important part is the manifest. The add-on manifest is a JSON object describing the add-on's capabilities.	JSON schema validation
Resource Access	Every resource is accessed at a certain endpoint where your add-on should respond with proper data.	Endpoint validation and sanitization

Addon Communication Flow:



Security Benefits:

Stremio's addon system was also created with the user's security in mind. The addons do not run any code locally, so they pose no risks to your

device. This architecture eliminates entire classes of security vulnerabilities associated with local code execution.

6.1.4 Database and Storage Layer

SQLite JSONB Performance Enhancement:

A big new feature is introduced in the SQLite 3.45.0 release – the SQLite JSONB. The aim of this feature is to speed up the JSON manipulation, since storing JSON as BLOB will save time normally spent on parsing the standard JSON saved as string.

Storage Architecture:

Data Category	Storage Format	Performance Benefit	Use Case
Content Metadata	JSONB BLOB	JSONB is a rewrite of the SQLite JSON functions that, depending on usage patterns, could be several times faster than the original JSON functions.	Movie/TV show details, cast information
User Preferences	Traditional SQL	Standard relational performance	Authentication, settings, watch history
Addon Manifests	JSONB BLOB	Binary format efficiency	Plugin configurations and capabilities
Cache Data	JSONB with TTL	Optimized read performance	API response caching

JSONB Performance Characteristics:

I compare JSON and JSON-B, explaining that JSON functions operate on text and convert it to binary, while JSON-B deals directly with the binary form, making it faster for operations.

Database Schema Design:

```sql
-- User management with traditional SQL
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Content metadata with JSONB optimization
CREATE TABLE content_metadata (
    id INTEGER PRIMARY KEY,
    tmdb_id INTEGER UNIQUE NOT NULL,
    content_type TEXT NOT NULL, -- 'movie' or 'tv'
    metadata BLOB, -- JSONB format
    cached_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    expires_at DATETIME NOT NULL
);

-- Addon configurations with JSONB
CREATE TABLE addons (
    id INTEGER PRIMARY KEY,
    url TEXT UNIQUE NOT NULL,
    manifest BLOB, -- JSONB format
    enabled BOOLEAN DEFAULT TRUE,
    installed_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Watch history with mixed approach
CREATE TABLE watch_history (
    id INTEGER PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    content_id INTEGER REFERENCES content_metadata(id),
    progress_seconds INTEGER DEFAULT 0,
    metadata BLOB, -- JSONB for additional data
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Performance Optimization Strategy:

The function will operate the same in either case, except that it will run faster when the input is JSONB, since it does not need to run the JSON parser. Most SQL functions that return JSON text have a corresponding function that returns the equivalent JSONB.

6.1.5 Video Player Integration Component

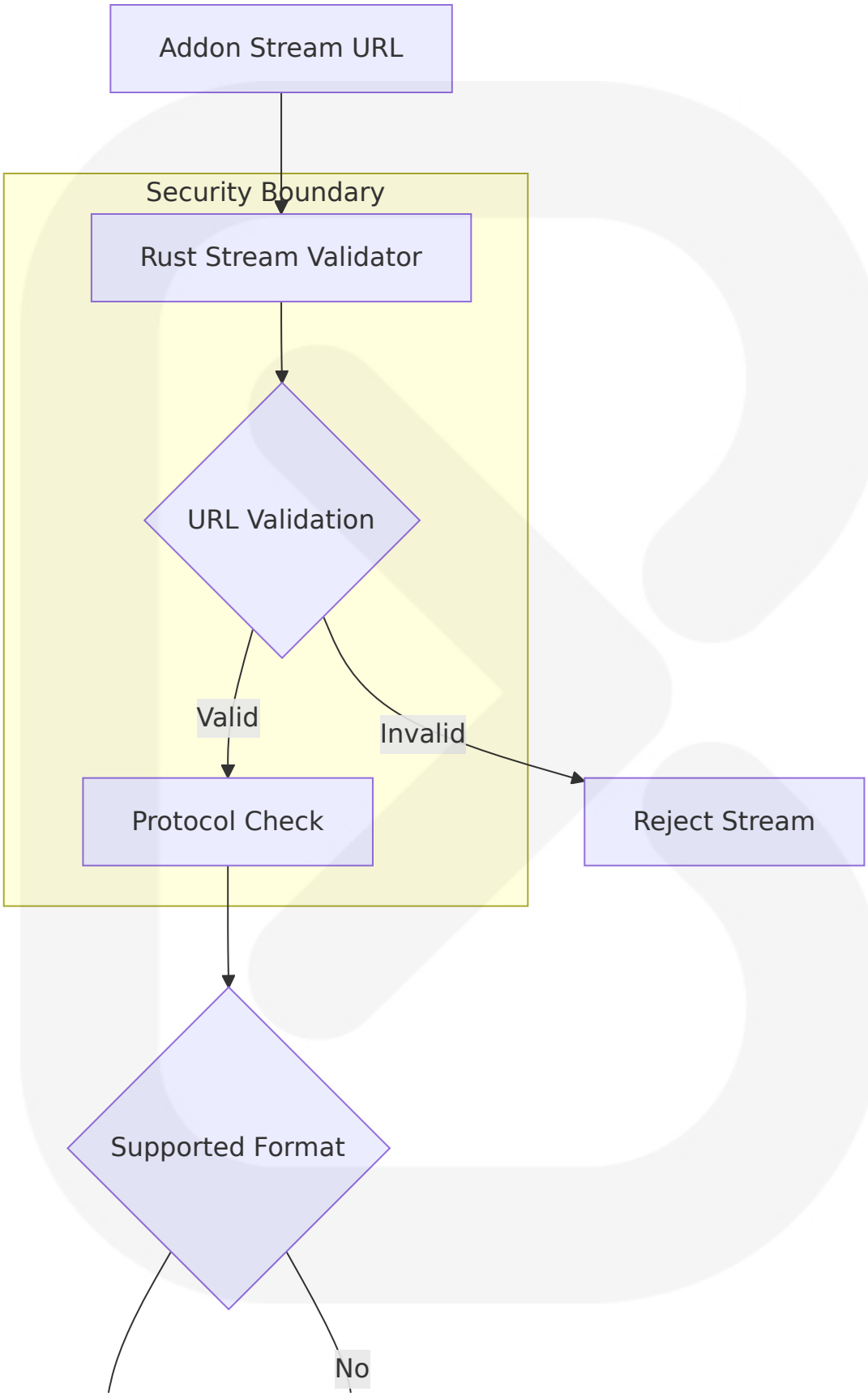
WebView-Based Player Architecture:

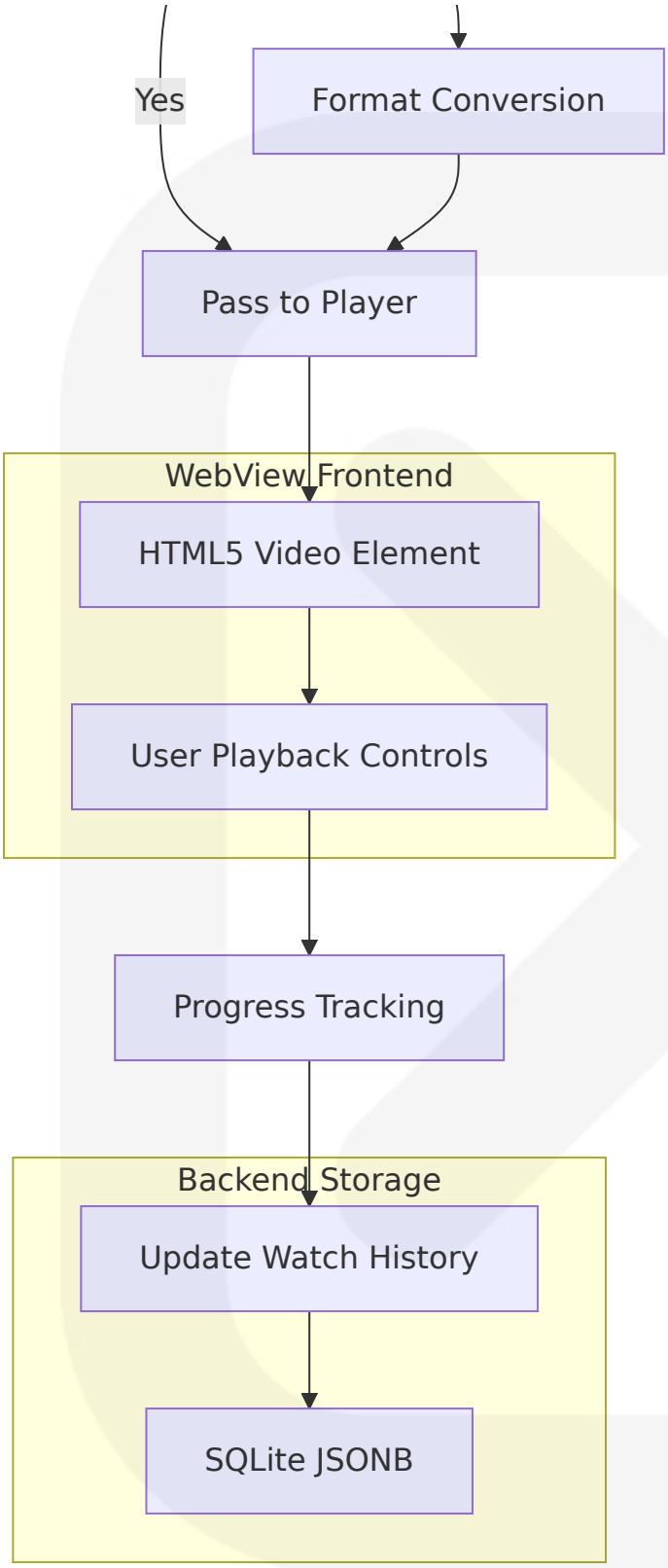
The video player component integrates with the WebView frontend while maintaining secure communication with the Rust backend for stream URL validation and processing.

Player Component Design:

Layer	Technology	Responsibility	Security Considerations
Player Interface	HTML5 Video + JavaScript	Video rendering, user controls	Content Security Policy enforcement
Stream Validator	Rust Backend	URL sanitization, format validation	Input validation, protocol checking
Casting Integration	Tauri Plugins	External device communication	Network permission management
Subtitle Handler	WebVTT/SRT Parser	Caption processing and display	File format validation

Stream Processing Flow:


Addon Stream URL
Security Boundary
Rust Stream Validator
URL Validation
Valid
Invalid
Protocol Check
Reject Stream
Supported Format
No



Casting and External Device Support:

The casting functionality leverages Tauri's plugin system to provide secure access to network protocols while maintaining the application's security boundaries.

6.2 COMPONENT INTERACTION PATTERNS

6.2.1 Inter-Process Communication (IPC) Design

Tauri IPC Architecture:

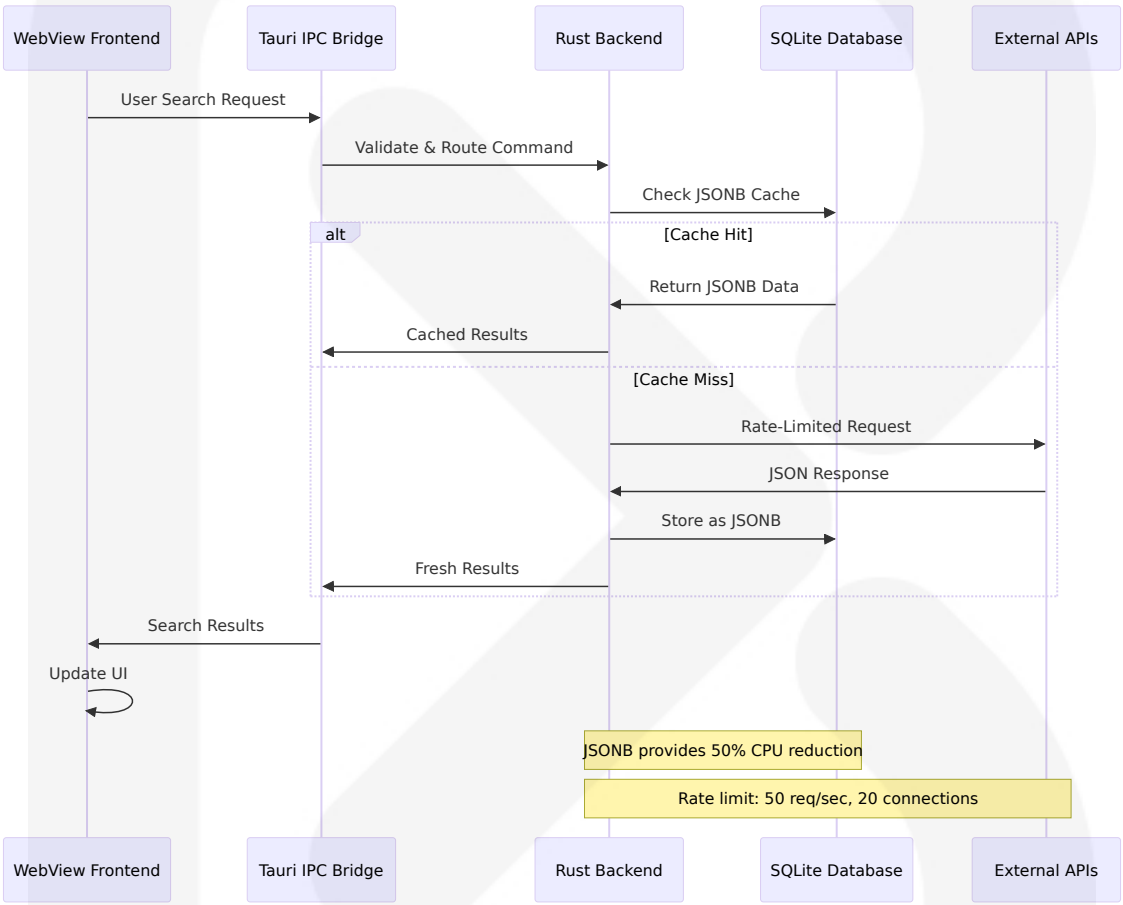
The application uses Tauri's secure IPC system for all communication between the WebView frontend and Rust backend, implementing asynchronous message passing with JSON serialization.

Command Pattern Implementation:

```
// Example Tauri command structure
#[tauri::command]
async fn search_content(
    query: String,
    filters: SearchFilters,
    state: tauri::State<'_, AppState>
) -> Result<SearchResults, SearchError> {
    // Rate limiting check
    state.rate_limiter.check_limit().await?;

    // Cache check with JSONB
    if let Some(cached) =
state.cache.get_search_results(&query).await? {
        return Ok(cached);
    }

    // TMDB API call with proper attribution
    let results = state.tmdb_client.search(&query, &filters).await?;
```

```
    // Store in JSONB format for performance
    state.cache.store_search_results(&query, &results).await?;

    Ok(results)
}
```

Event-Driven Communication:



6.2.2 Data Flow Architecture

Content Discovery Data Flow:

The content discovery system implements a multi-layered caching strategy with intelligent cache invalidation and TMDB API integration.

Cache Hierarchy:

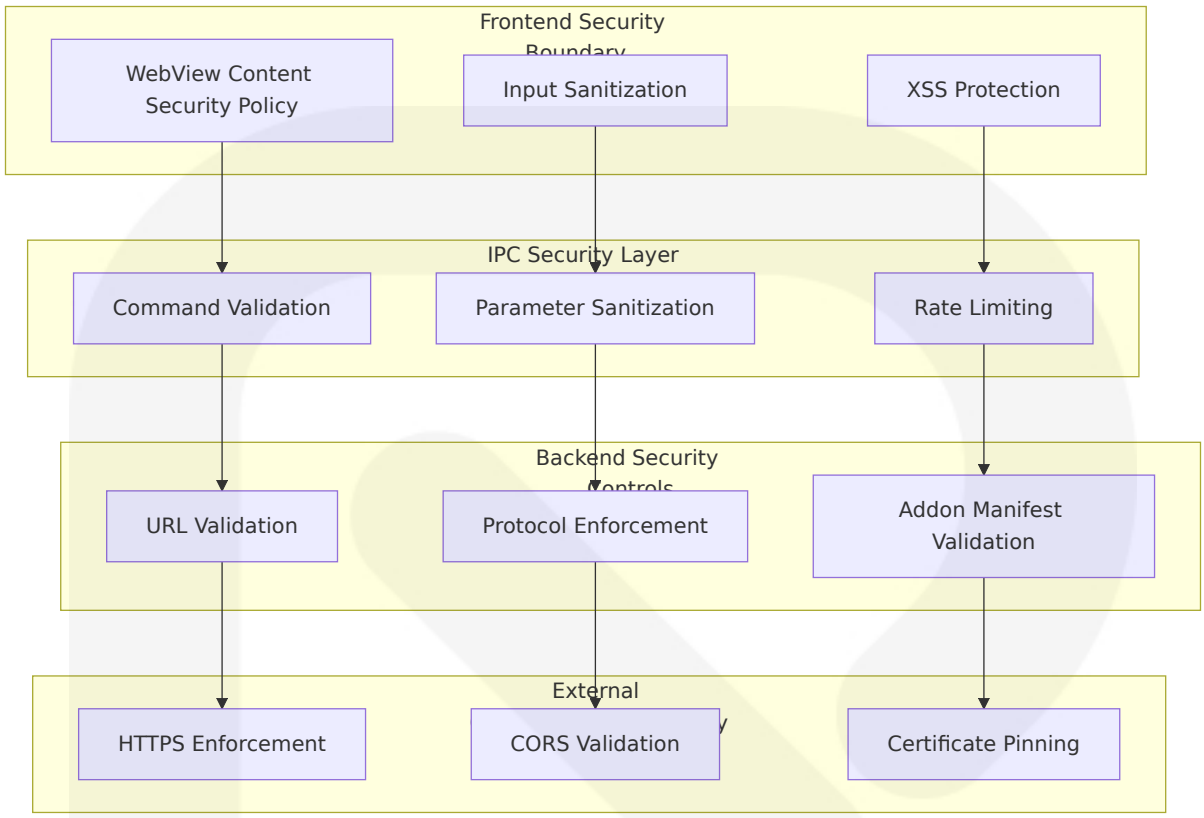
Cache Level	Storage Type	TTL	Purpose
Memory Cache	In-process HashMap	5 minutes	Frequently accessed data
JSONB Cache	SQLite BLOB	24 hours (search), 7 days (metadata)	Persistent local storage
Image Cache	File system	30 days	Poster and backdrop images
Addon Response Cache	SQLite JSONB	1 hour	Addon manifest and stream data

Performance Optimization Flow:

Thus, we expect that using JSONB will have better performance because the engine will only need to convert to and from the text format when the data is inserted and when it's output to the user, instead of during every operation.

6.2.3 Security Component Integration

Multi-Layer Security Architecture:



Addon Security Validation:

The addon system implements comprehensive security validation to ensure safe remote execution without local code risks.

Validation Pipeline:

1. **URL Security Check**: Enforce HTTPS for remote addons (localhost exceptions allowed)
2. **CORS Validation**: Verify proper cross-origin headers
3. **Manifest Schema Validation**: Ensure addon capabilities are properly declared
4. **Content Sanitization**: Validate all stream URLs and metadata
5. **Rate Limiting**: Prevent abuse of addon endpoints

6.3 SCALABILITY AND PERFORMANCE DESIGN

6.3.1 Performance Optimization Strategies

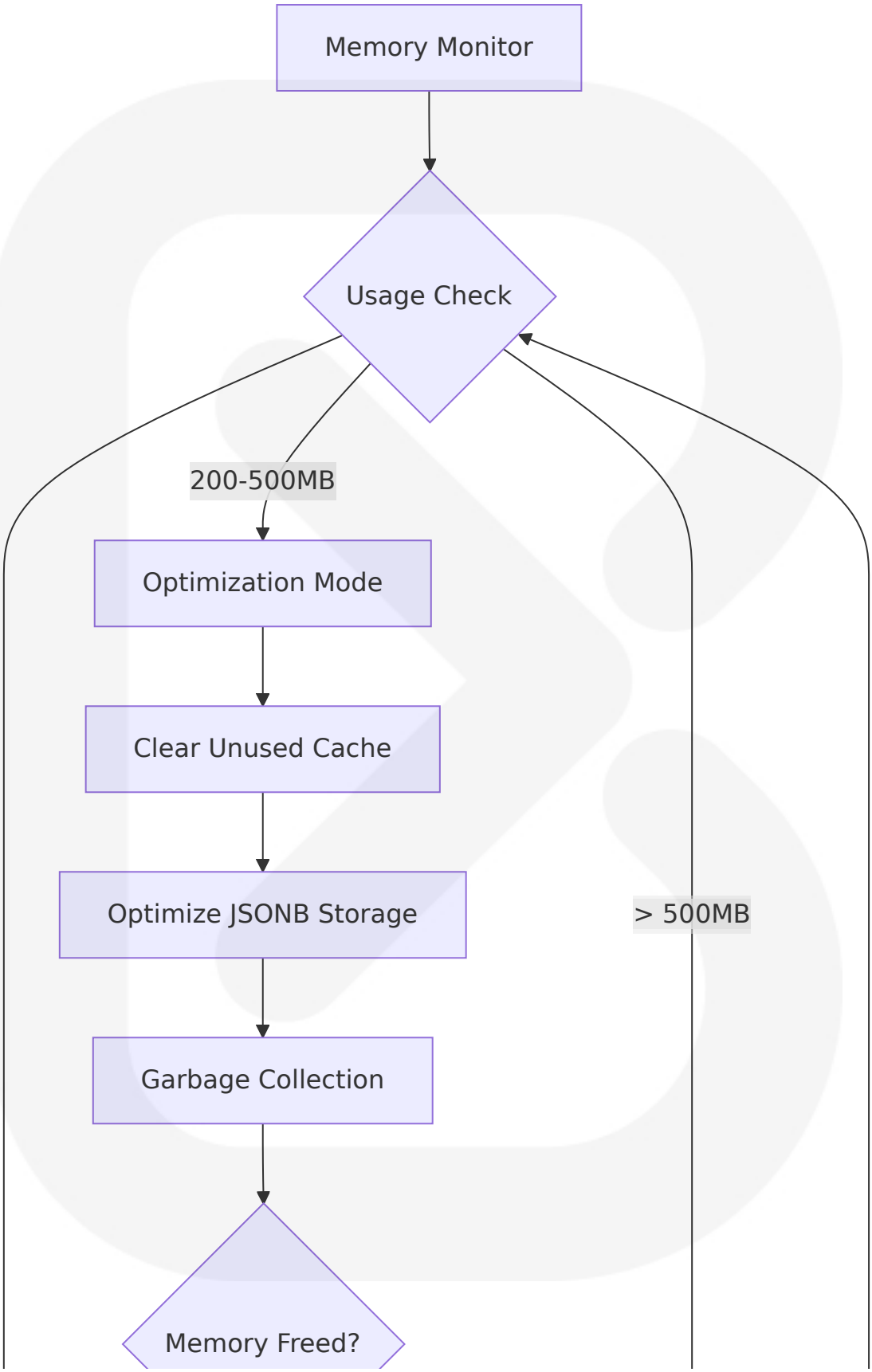
SQLite JSONB Performance Benefits:

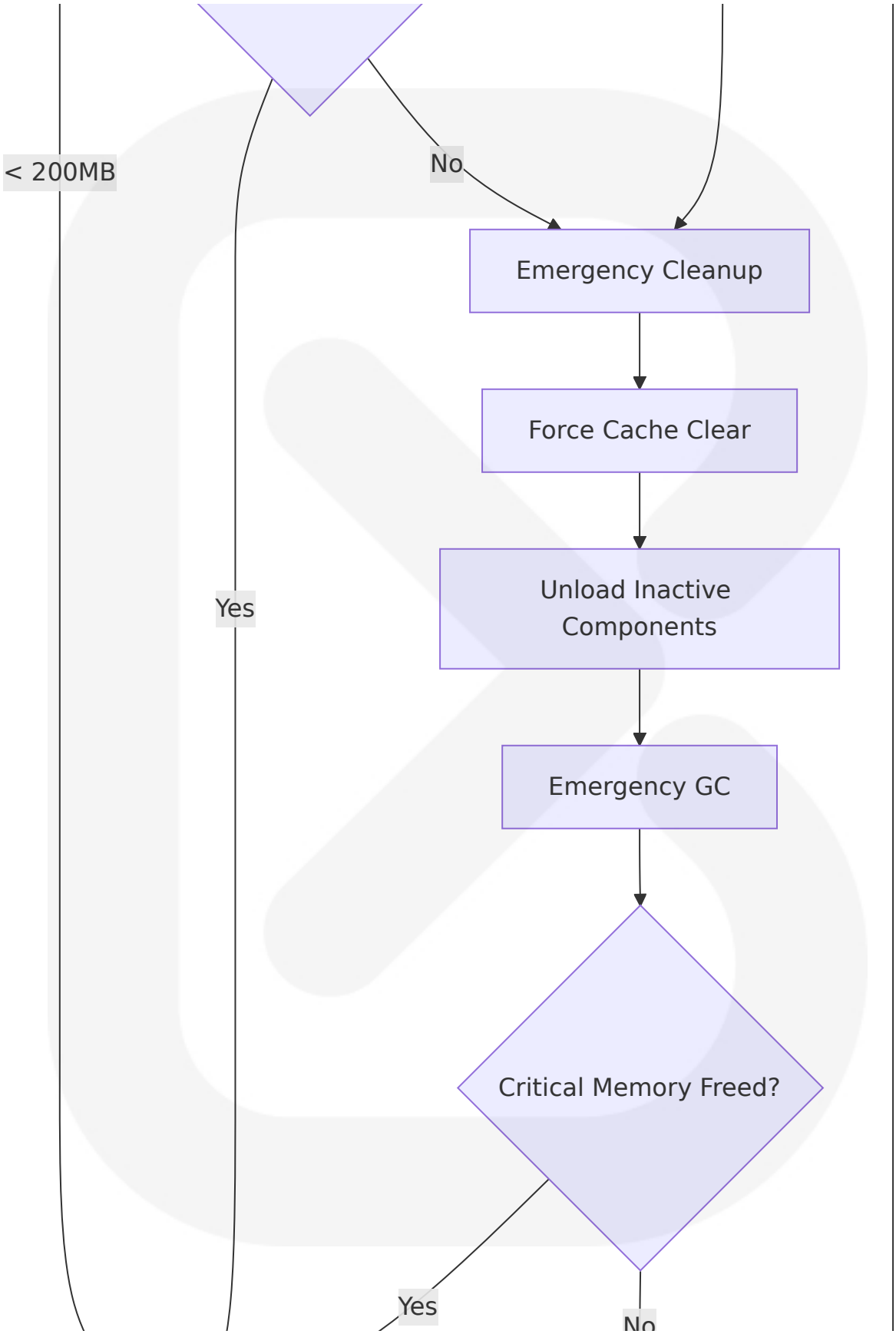
The advantage of JSONB over ordinary text RFC 8259 JSON is that JSONB is both slightly smaller (by between 5% and 10% in most cases) and can be processed in less than half the number of CPU cycles.

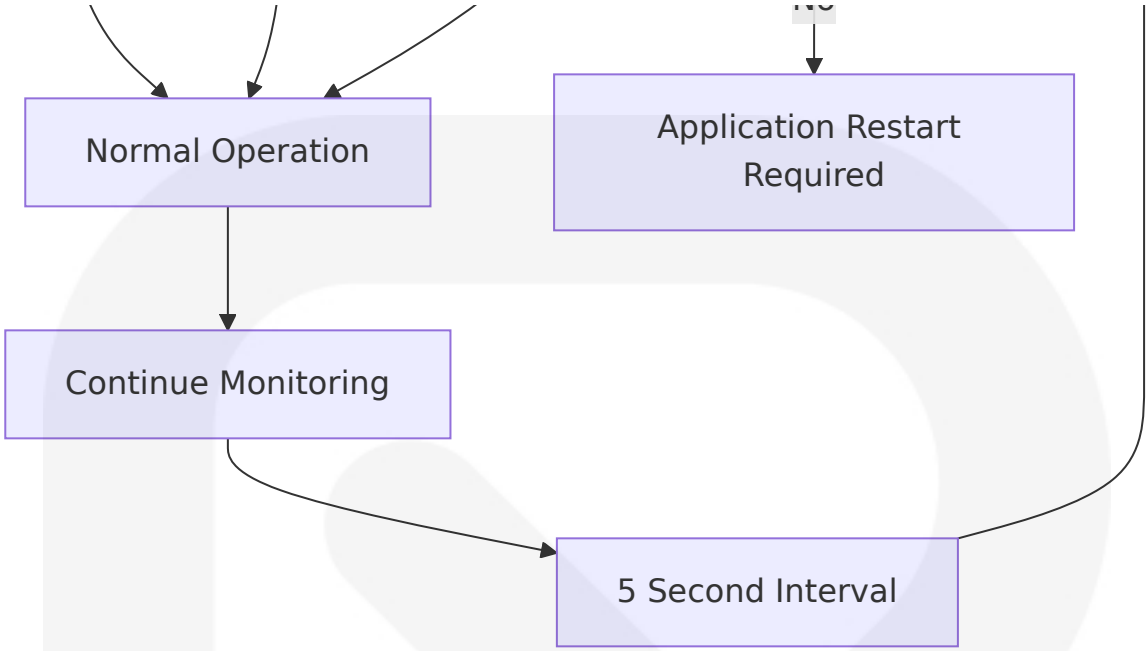
Performance Metrics and Targets:

Component	Performance Target	Measurement Method	Optimization Strategy
Application Startup	< 3 seconds	Automated timing	Lazy loading, optimized initialization
Search Response	< 1 second	User interaction tracking	JSONB caching, intelligent prefetching
IPC Communication	< 100ms	Built-in Tauri metrics	Efficient serialization, command batching
Video Stream Start	< 5 seconds	Media player integration	Stream preloading, format optimization

Memory Management Strategy:


Memory Monitor
Usage Check
200-500MB
Optimization Mode
Clear Unused Cache
Optimize JSONB Storage
Garbage Collection
Memory Freed?
> 500MB


< 200MB
No
Emergency Cleanup
Force Cache Clear
Yes
Unload Inactive Components
Emergency GC
Critical Memory Freed?
Yes
No



6.3.2 Concurrent Processing Design

Async Runtime Architecture:

The application leverages Tokio's async runtime for efficient concurrent processing of I/O-bound operations like API requests and database queries.

Concurrency Patterns:

Operation Type	Concurrency Strategy	Performance Benefit
API Requests	Connection pooling with 20 max connections	Optimal TMDB rate limit utilization
Database Operations	SQLite WAL mode with concurrent reads	Improved read performance
Image Loading	Parallel download with semaphore limiting	Faster UI loading
Addon Communication	Async request batching	Reduced latency

Resource Management:

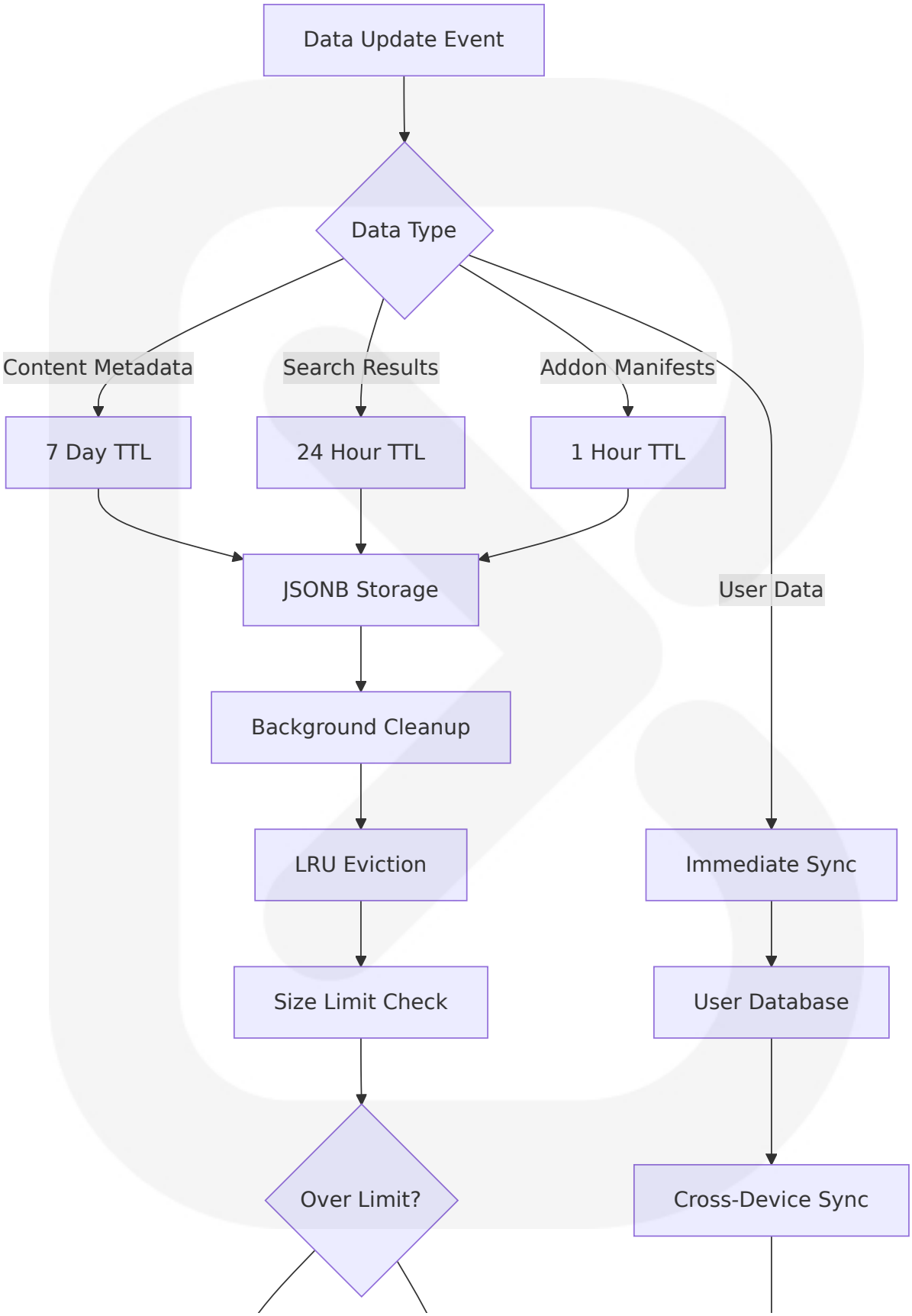
```rust
// Example concurrent processing structure
pub struct ContentDiscoveryService {
    tmdb_client: Arc<TmdbClient>,
    cache: Arc<JsonbCache>,
    rate_limiter: Arc<RateLimiter>,
    semaphore: Arc<Semaphore>, // Limit concurrent operations
}

impl ContentDiscoveryService {
    pub async fn batch_search(
        &self,
        queries: Vec<String>
    ) -> Result<Vec<SearchResult>, ServiceError> {
        let futures = queries.into_iter().map(|query| {
            let client = Arc::clone(&self.tmdb_client);
            let cache = Arc::clone(&self.cache);
            let limiter = Arc::clone(&self.rate_limiter);
            let permit = Arc::clone(&self.semaphore);

            async move {
                let _permit = permit.acquire().await?;
                limiter.check_limit().await?;

                if let Some(cached) = cache.get(&query).await? {
                    return Ok(cached);
                }

                let result = client.search(&query).await?;
                cache.store_jsonb(&query, &result).await?;
                Ok(result)
            }
        });

        try_join_all(futures).await
    }
}
```

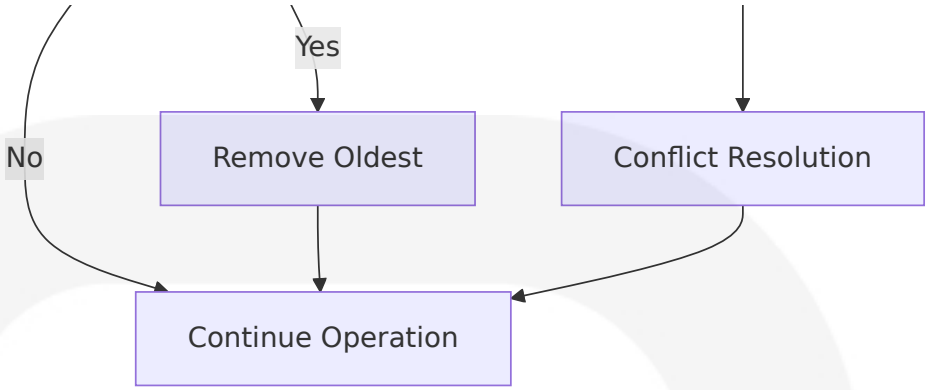
6.3.3 Caching and Storage Optimization

JSONB Storage Strategy:

The JSONB rewrite changes the internal-use binary representation of JSON into a contiguous byte array that can read or written as an SQL BLOB. This allows the internal-use representation of JSON to potentially be saved to the database, in place of JSON text, eliminating the overhead of steps 1 and 3.

Cache Invalidation Strategy:


Data Update Event
Data Type
Content Metadata
Search Results
Addon Manifests
7 Day TTL
24 Hour TTL
1 Hour TTL
JSONB Storage
User Data
Background Cleanup
LRU Eviction
Immediate Sync
Size Limit Check
User Database
Over Limit?
Cross-Device Sync



Storage Efficiency Metrics:

- **JSONB Space Savings**: 5-10% reduction compared to text JSON
- **CPU Performance**: 50% reduction in processing cycles
- **Cache Hit Ratio Target**: > 80% for frequently accessed content
- **Database Size Management**: Automatic cleanup when exceeding 1GB

This comprehensive system components design ensures optimal performance, security, and scalability while leveraging the latest technologies like Tauri 2.0 and SQLite JSONB for maximum efficiency in the Stremio clone desktop application.

6.1 CORE SERVICES ARCHITECTURE

6.1.1 Architecture Applicability Assessment

Core Services Architecture is not applicable for this system due to the fundamental architectural characteristics of the Stremio Clone Desktop Application.

The application is designed as a single unified desktop application built with Tauri 2.0, where the frontend is written in web technologies and runs inside the operating system WebView, communicating with the application core written mostly in Rust. This represents a monolithic architecture

model where a single codebase is used to perform multiple functions in an application.

6.1.2 Architectural Rationale

Monolithic Desktop Application Design:

The Stremio Clone follows a monolithic application pattern as a single unified software application that is self-contained and independent from other applications. This architectural choice is justified by several factors:

Design Factor	Justification	Benefit
Desktop Application Context	Traditional desktop applications, especially legacy software suites like older versions of Microsoft Office, were commonly built with a monolithic architecture	Simplified deployment and maintenance
Single User Environment	Desktop applications typically serve individual users rather than distributed systems	Eliminates need for service discovery and load balancing
Performance Requirements	Interactions are typically more efficient since all communication is local, and monolithic applications can deliver consistent performance with no network latency or communication overhead between different services	Optimal user experience for media streaming

Tauri Framework Characteristics:

Tauri applications are very small because they use the OS's webview and do not ship a runtime since the final binary is compiled from Rust. The framework provides:

- **Process Isolation**: While maintaining a monolithic application structure, Tauri employs a multi-process architecture for security

- **Unified Deployment**: The entire application is packaged and deployed as a single unit, making application integration easier
- **Local Operations**: Since there's a single component, all operations are local

6.1.3 Alternative Architecture Considerations

Why Microservices Are Not Suitable:

One of the big selling points of microservice architecture is scalability - the ability to scale out any specific part of your application independently. In theory you can start 5, 10 or 100 instances of a microservice. Is this applicable in a desktop application at all?

Desktop Application Constraints:

Constraint	Impact on Service Architecture	Monolithic Advantage
Single User Context	No need for horizontal scaling across multiple users	Many applications, when they need to scale beyond a single instance, can do so through cloning that entire instance rather than separating into discrete services
Local Resource Limits	Desktop hardware constraints make service distribution unnecessary	Efficient resource utilization within single process
Network Security	Desktop applications on client machines with security policies (firewalls, etc.) would face challenges exposing ports for inter-service communication	No internal network communication required

6.1.4 Monolithic Architecture Benefits for This System

Development and Maintenance Advantages:

Building a monolithic application is often faster and more straightforward, especially for smaller projects with well-defined requirements. This streamlined approach can accelerate initial development time.

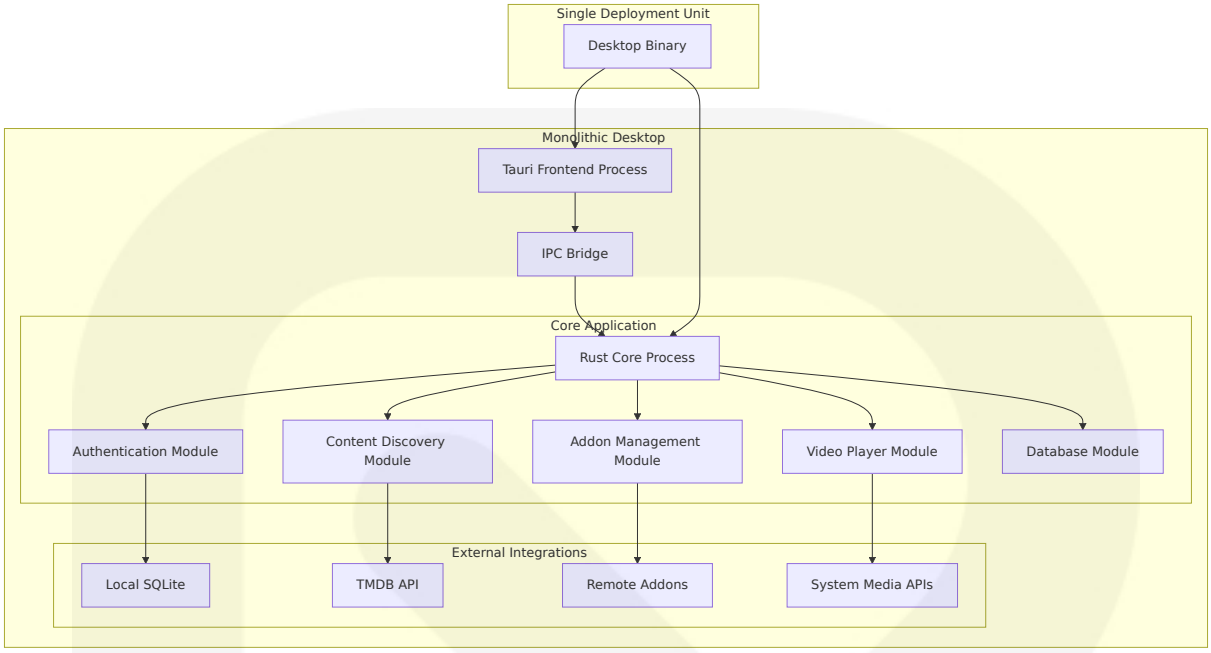
Operational Benefits:

Benefit Category	Specific Advantage	Implementation Impact
Simplified Debugging	Debugging is straightforward since you're working within a single codebase, and tracing issues can be more straightforward compared to distributed architectures	Faster issue resolution
Consistent Performance	All components run within the same process, delivering consistent performance	Optimal media streaming experience
Unified Testing	Testing and debugging operations are considerably less intensive with monolithic architectures, enacted from a central logging system	Comprehensive quality assurance

6.1.5 System Component Integration

Internal Component Architecture:

While the application maintains a monolithic structure, it implements internal modularity through Tauri's plugin system and Rust's module organization:



Modular Monolith Benefits:

The monolithic architecture can increase maintainability and team autonomy by modularizing the monolith, organizing subdomains into vertical slices consisting of presentation, business and persistence layers.

6.1.6 Scalability Through Monolithic Design

Resource Scaling Strategy:

For desktop applications, scalability focuses on efficient resource utilization rather than horizontal distribution:

Scaling Dimension	Monolithic Approach	Implementation
Performance Scaling	Optimize single-process efficiency	Rust's zero-cost abstractions, efficient memory management
Feature Scaling	Modular code organization within monolith	Plugin system for extensibility
Data Scaling	Local database optimization	SQLite with JSONB for performance

Scaling Dimension	Monolithic Approach	Implementation
User Scaling	Per-installation optimization	Individual user customization and caching

Conclusion:

The Stremio Clone Desktop Application's architecture is fundamentally incompatible with distributed services patterns. Monolithic applications are still a good choice for applications with small teams and little complexity, though once it becomes too complex, you can consider refactoring into microservices or distributed applications. For this desktop media center application, the monolithic approach provides optimal performance, simplified deployment, and efficient resource utilization while meeting all functional requirements through internal modularity and external API integrations.

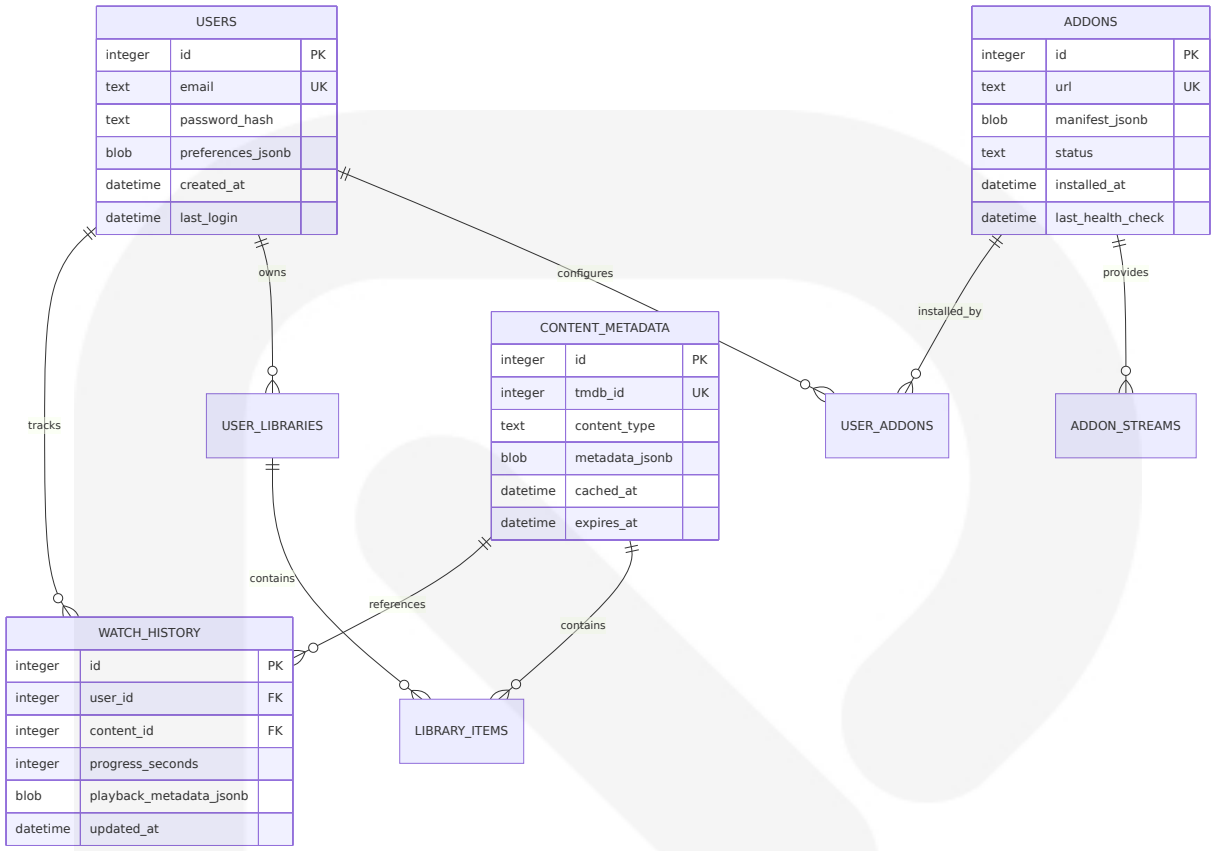
6.2 DATABASE DESIGN

6.2.1 SCHEMA DESIGN

6.2.1.1 Entity Relationships

The Stremio Clone Desktop Application requires a comprehensive database design to support user management, content discovery, addon management, and media playback tracking. The system leverages SQLite's JSONB format, which is both slightly smaller (by between 5% and 10% in most cases) and can be processed in less than half the number of CPU cycles compared to traditional JSON storage.

Core Entity Relationships:



6.2.1.2 Data Models and Structures

Primary Data Models:

Entity	Storage Strategy	JSONB Usage	Performance Benefit
Content Metadata	Hybrid relational + JSONB	Movie/TV details, cast, crew	50% CPU reduction for JSON operations
User Preferences	JSONB blob	Settings, UI preferences, filters	Flexible schema evolution
Addon Manifests	JSONB blob	Plugin capabilities, endpoints	5-10% space savings over text JSON
Watch History	Mixed approach	Progress tracking with metadata	Optimized read performance

SQLite Schema Implementation:

```
-- Users table with JSONB preferences
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    preferences_jsonb BLOB, -- JSONB format for user settings
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_login DATETIME,
    sync_token TEXT -- For cross-device synchronization
);

-- Content metadata with JSONB optimization
CREATE TABLE content_metadata (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    tmdb_id INTEGER UNIQUE NOT NULL,
    content_type TEXT NOT NULL CHECK (content_type IN ('movie', 'tv',
'episode')),
    title TEXT NOT NULL,
    metadata_jsonb BLOB, -- JSONB format for rich metadata
    cached_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    expires_at DATETIME NOT NULL,
    search_vector TEXT -- For full-text search
);

-- Addon management with JSONB manifests
CREATE TABLE addons (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    url TEXT UNIQUE NOT NULL,
    name TEXT NOT NULL,
    manifest_jsonb BLOB, -- JSONB format for addon capabilities
    status TEXT DEFAULT 'active' CHECK (status IN ('active',
'disabled', 'error')),
    installed_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_health_check DATETIME,
    health_status TEXT DEFAULT 'unknown'
);

-- User-specific addon configurations
CREATE TABLE user_addons (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    addon_id INTEGER REFERENCES addons(id) ON DELETE CASCADE,
    config_jsonb BLOB, -- JSONB format for user-specific settings
```

```
    enabled BOOLEAN DEFAULT TRUE,
    installed_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, addon_id)
);

-- Watch history with JSONB metadata
CREATE TABLE watch_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    content_id INTEGER REFERENCES content_metadata(id) ON DELETE
CASCADE,
    progress_seconds INTEGER DEFAULT 0,
    total_duration_seconds INTEGER,
    playback_metadata_jsonb BLOB, -- JSONB for quality, subtitles,
etc.
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, content_id)
);

-- User libraries (watchlists, favorites)
CREATE TABLE user_libraries (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    name TEXT NOT NULL,
    type TEXT NOT NULL CHECK (type IN ('watchlist', 'favorites',
'custom')),
    metadata_jsonb BLOB, -- JSONB for library settings
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, name)
);

-- Library items
CREATE TABLE library_items (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    library_id INTEGER REFERENCES user_libraries(id) ON DELETE
CASCADE,
    content_id INTEGER REFERENCES content_metadata(id) ON DELETE
CASCADE,
    added_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    notes TEXT,
    UNIQUE(library_id, content_id)
);
```

```
-- Cache management table
CREATE TABLE cache_metadata (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    cache_key TEXT UNIQUE NOT NULL,
    cache_type TEXT NOT NULL,
    data_jsonb BLOB, -- JSONB format for cached data
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    expires_at DATETIME NOT NULL,
    access_count INTEGER DEFAULT 0,
    last_accessed DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

6.2.1.3 Indexing Strategy

Performance-Optimized Indexes:

Index Type	Purpose	Performance Impact	JSONB Integration
Primary Keys	Unique identification	O(log n) lookups	Standard B-tree indexes
Foreign Keys	Referential integrity	Join optimization	Automatic index creation
JSONB Indexes	JSON field queries	Faster JSON operations without parsing	Expression-based indexes
Full-Text Search	Content discovery	Sub-second search	FTS5 virtual tables

Index Implementation:

```
-- Standard indexes for foreign keys and lookups
CREATE INDEX idx_content_metadata_tmdb_id ON
content_metadata(tmdb_id);
CREATE INDEX idx_content_metadata_type ON
content_metadata(content_type);
CREATE INDEX idx_content_metadata_expires ON
content_metadata(expires_at);

-- JSONB expression indexes for common queries
```

```
CREATE INDEX idx_content_metadata_title ON
content_metadata(json_extract(metadata_jsonb, '$.title'));
CREATE INDEX idx_content_metadata_year ON
content_metadata(json_extract(metadata_jsonb, '$.release_date'));
CREATE INDEX idx_content_metadata_rating ON
content_metadata(json_extract(metadata_jsonb, '$.vote_average'));

-- Watch history optimization
CREATE INDEX idx_watch_history_user_updated ON watch_history(user_id,
updated_at DESC);
CREATE INDEX idx_watch_history_progress ON watch_history(user_id,
progress_seconds)
    WHERE progress_seconds > 0;

-- Addon management indexes
CREATE INDEX idx_addons_status ON addons(status) WHERE status =
'active';
CREATE INDEX idx_user_addons_enabled ON user_addons(user_id, enabled)
WHERE enabled = TRUE;

-- Cache management indexes
CREATE INDEX idx_cache_expires ON cache_metadata(expires_at);
CREATE INDEX idx_cache_type_key ON cache_metadata(cache_type,
cache_key);
CREATE INDEX idx_cache_access ON cache_metadata(last_accessed DESC);

-- Full-text search for content discovery
CREATE VIRTUAL TABLE content_search USING fts5(
    title,
    overview,
    cast_names,
    genre_names,
    content=content_metadata,
    content_rowid=id
);
```

6.2.1.4 Partitioning Approach

SQLite Partitioning Strategy:

SQLite does not support traditional table partitioning, but the application implements logical partitioning strategies for optimal performance:

Partition Strategy	Implementation	Performance Benefit
Time-based Partitioning	Separate tables by date ranges	Improved query performance for recent data
User-based Partitioning	Per-user data isolation	Enhanced privacy and performance
Content-type Partitioning	Separate movie/TV metadata	Optimized schema per content type

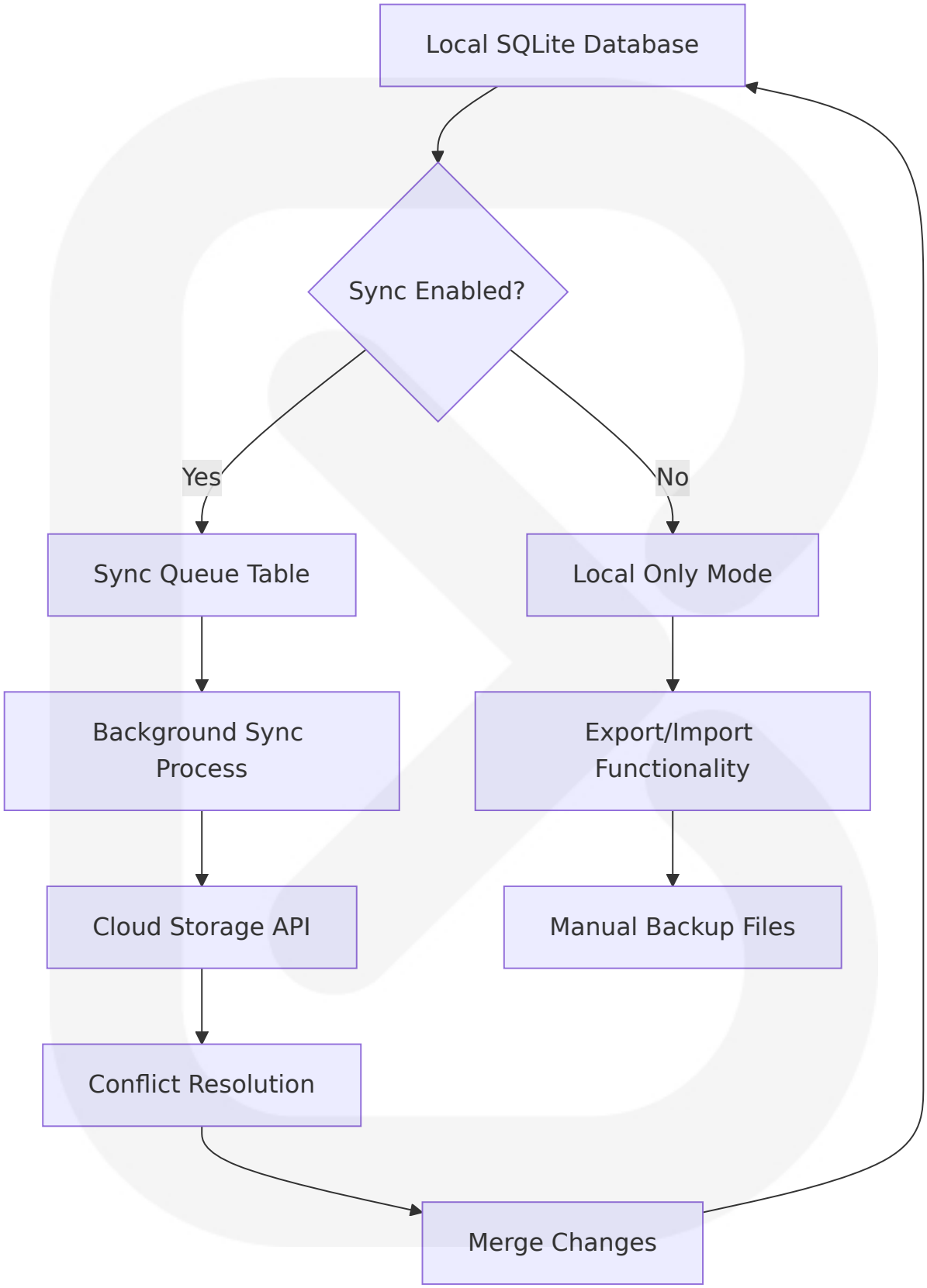
Logical Partitioning Implementation:

```
-- Time-based partitioning for watch history
CREATE TABLE watch_history_current AS SELECT * FROM watch_history
WHERE 1=0;
CREATE TABLE watch_history_archive AS SELECT * FROM watch_history
WHERE 1=0;

-- Trigger for automatic partitioning
CREATE TRIGGER partition_watch_history
AFTER INSERT ON watch_history
WHEN NEW.updated_at < date('now', '-90 days')
BEGIN
    INSERT INTO watch_history_archive SELECT * FROM watch_history
WHERE id = NEW.id;
    DELETE FROM watch_history WHERE id = NEW.id;
END;

-- Content-type specific optimization
CREATE VIEW movies_metadata AS
SELECT * FROM content_metadata WHERE content_type = 'movie';

CREATE VIEW tv_metadata AS
SELECT * FROM content_metadata WHERE content_type = 'tv';
```

6.2.1.5 Replication Configuration

Local-First Replication Strategy:

The application implements a local-first approach with optional cloud synchronization:


Local SQLite Database
Sync Enabled?
Yes
No
Sync Queue Table
Local Only Mode
Background Sync Process
Export/Import Functionality
Cloud Storage API
Manual Backup Files
Conflict Resolution
Merge Changes

Synchronization Schema:

```
-- Sync queue for cloud synchronization
CREATE TABLE sync_queue (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    table_name TEXT NOT NULL,
    record_id INTEGER NOT NULL,
    operation TEXT NOT NULL CHECK (operation IN ('INSERT', 'UPDATE',
'DELETE')),
    data_jsonb BLOB, -- JSONB format for change data
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    synced_at DATETIME,
    retry_count INTEGER DEFAULT 0,
    error_message TEXT
);

-- Sync metadata tracking
CREATE TABLE sync_metadata (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id),
    last_sync_timestamp DATETIME,
    sync_token TEXT,
    device_id TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

6.2.1.6 Backup Architecture

Multi-Layer Backup Strategy:

Backup Type	Frequency	Storage Location	Recovery Time
WAL Checkpoints	Automatic	Local filesystem	Immediate
Full Database Backup	Daily	Local + Cloud	< 5 minutes
Incremental Sync	Real-time	Cloud storage	< 1 minute
Export Snapshots	On-demand	User-specified	Manual restore

Backup Implementation:

```
-- Backup metadata tracking
CREATE TABLE backup_metadata (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    backup_type TEXT NOT NULL CHECK (backup_type IN ('full',
'incremental', 'export')),
    file_path TEXT,
    file_size INTEGER,
    checksum TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    status TEXT DEFAULT 'completed'
);

-- Database integrity checks
PRAGMA integrity_check;
PRAGMA foreign_key_check;
PRAGMA quick_check;
```

6.2.2 DATA MANAGEMENT

6.2.2.1 Migration Procedures

SQLite Migration Strategy with Tauri Integration:

The Tauri SQL plugin supports database migrations, allowing you to manage database schema evolution over time. Migrations are defined in Rust using the Migration struct. Each migration should include a unique version number, a description, the SQL to be executed, and the type of migration (Up or Down).

Migration Implementation:

```
use tauri_plugin_sql::{Migration, MigrationKind};

// Migration definitions for the Stremio Clone
let migrations = vec![
    // Initial schema creation
```

```
    Migration {
        version: 1,
        description: "Create initial user and content tables",
        sql: r#"
            CREATE TABLE users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                email TEXT UNIQUE NOT NULL,
                password_hash TEXT NOT NULL,
                preferences_jsonb BLOB,
                created_at DATETIME DEFAULT CURRENT_TIMESTAMP
            );

            CREATE TABLE content_metadata (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                tmdb_id INTEGER UNIQUE NOT NULL,
                content_type TEXT NOT NULL,
                metadata_jsonb BLOB,
                cached_at DATETIME DEFAULT CURRENT_TIMESTAMP,
                expires_at DATETIME NOT NULL
            );
        "#,
        kind: MigrationKind::Up,
    },

    // JSONB optimization migration
    Migration {
        version: 2,
        description: "Optimize existing JSON data to JSONB format",
        sql: r#"
            -- Convert existing JSON text to JSONB binary format
            UPDATE content_metadata
            SET metadata_jsonb = jsonb(metadata_jsonb)
            WHERE metadata_jsonb IS NOT NULL;

            -- Add indexes for JSONB fields
            CREATE INDEX idx_content_jsonb_title
            ON content_metadata(jsonb_extract(metadata_jsonb,
'$.title'));
        "#,
        kind: MigrationKind::Up,
    },

    // Addon system migration
```

```
    Migration {
        version: 3,
        description: "Add addon management tables",
        sql: r#"
            CREATE TABLE addons (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                url TEXT UNIQUE NOT NULL,
                manifest_jsonb BLOB,
                status TEXT DEFAULT 'active',
                installed_at DATETIME DEFAULT CURRENT_TIMESTAMP
            );

            CREATE TABLE user_addons (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                user_id INTEGER REFERENCES users(id),
                addon_id INTEGER REFERENCES addons(id),
                config_jsonb BLOB,
                enabled BOOLEAN DEFAULT TRUE
            );
        "#,
        kind: MigrationKind::Up,
    }
];
```

Migration Registration:

```
// Register migrations with Tauri plugin
tauri::Builder::default()
    .plugin(
        tauri_plugin_sql::Builder::default()
            .add_migrations("sqlite:stremio_clone.db", migrations)
            .build()
    )
    .run(tauri::generate_context!())
    .expect("error while running Tauri application");
```

6.2.2.2 Versioning Strategy

Database Version Management:

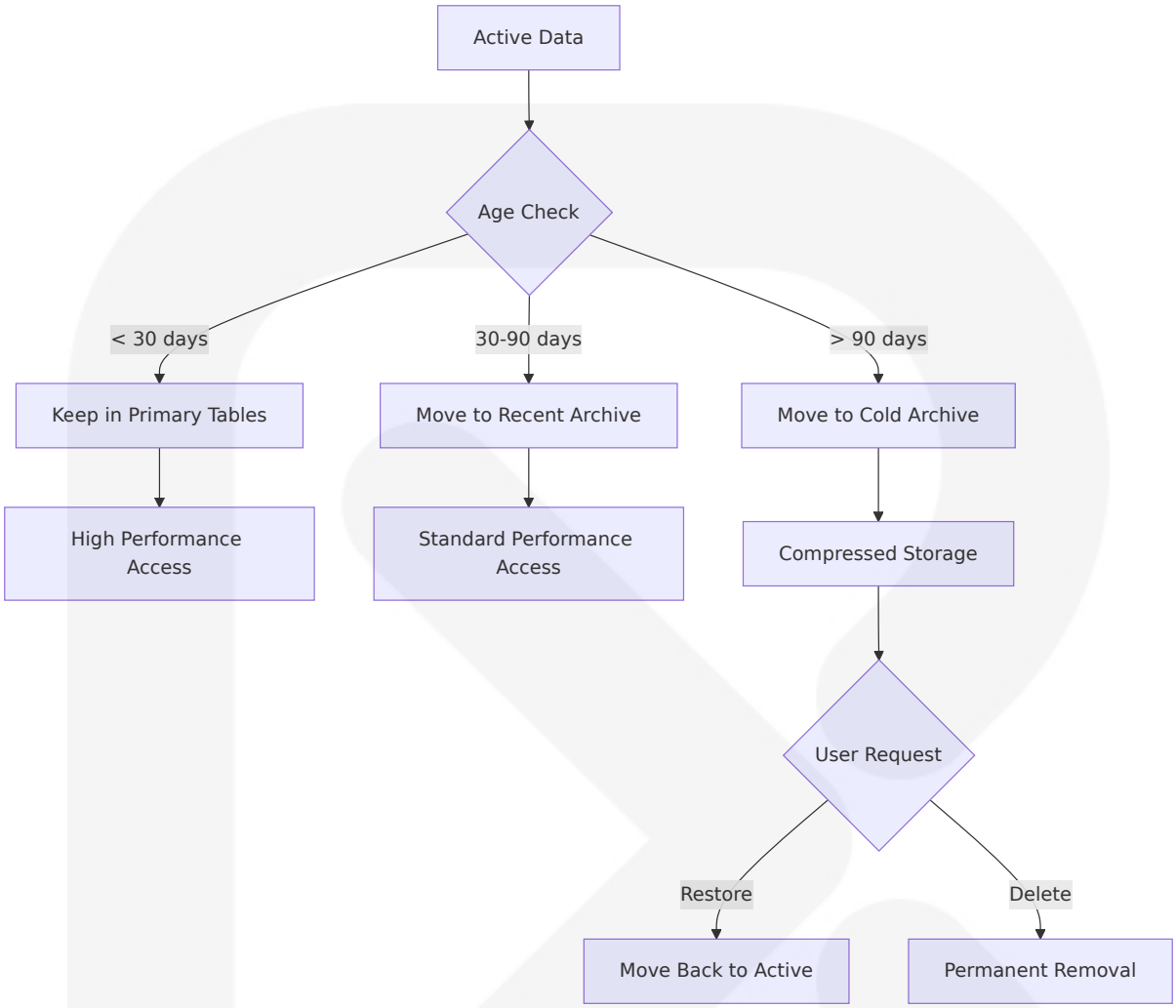
Version Component	Purpose	Implementation	Rollback Strategy
Schema Version	Track structural changes	Migration version numbers	Down migrations
Data Version	Track data format changes	Version metadata table	Data transformation scripts
Application Version	Compatibility tracking	App version in database	Version compatibility matrix

Version Tracking Schema:

```
-- Database version tracking
CREATE TABLE schema_versions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    version_number INTEGER UNIQUE NOT NULL,
    description TEXT NOT NULL,
    applied_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    rollback_sql TEXT,
    checksum TEXT
);

-- Application compatibility matrix
CREATE TABLE app_compatibility (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    app_version TEXT NOT NULL,
    min_schema_version INTEGER NOT NULL,
    max_schema_version INTEGER NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

6.2.2.3 Archival Policies

Data Lifecycle Management:



Archival Implementation:

```
-- Archival policy configuration
CREATE TABLE archival_policies (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    table_name TEXT NOT NULL,
    retention_days INTEGER NOT NULL,
    archive_action TEXT CHECK (archive_action IN ('move', 'compress',
'delete')),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Automated archival trigger
CREATE TRIGGER archive_old_cache
AFTER INSERT ON cache_metadata
WHEN NEW.expires_at < date('now', '-30 days')
```

```
BEGIN
    DELETE FROM cache_metadata WHERE expires_at < date('now', '-30
days');
END;

-- Watch history archival
CREATE TABLE watch_history_archive (
    id INTEGER PRIMARY KEY,
    user_id INTEGER,
    content_id INTEGER,
    progress_seconds INTEGER,
    archived_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    original_data_jsonb BLOB
);
```

6.2.2.4 Data Storage and Retrieval Mechanisms

JSONB-Optimized Storage Strategy:

JSON functions operate on text and convert it to binary, while JSON-B deals directly with the binary form, making it faster for operations. The application leverages this performance advantage through strategic JSONB usage.

Storage Optimization Patterns:

```
// Rust implementation for JSONB storage
use serde::{Deserialize, Serialize};
use sqlx::{Row, SqlitePool};

#[derive(Serialize, Deserialize)]
struct ContentMetadata {
    title: String,
    overview: String,
    release_date: String,
    vote_average: f64,
    genres: Vec<String>,
    cast: Vec<CastMember>,
}
```

```rust
// Store content metadata using JSONB
async fn store_content_metadata(
    pool: &SqlitePool,
    tmdb_id: i64,
    metadata: &ContentMetadata,
) -> Result<(), sqlx::Error> {
    let metadata_json = serde_json::to_string(metadata)?;

    sqlx::query!(
        r#"
        INSERT OR REPLACE INTO content_metadata
        (tmdb_id, content_type, metadata_jsonb, expires_at)
        VALUES (?, 'movie', jsonb(?), datetime('now', '+7 days'))
        "#,
        tmdb_id,
        metadata_json
    )
    .execute(pool)
    .await?;

    Ok(())
}

// Retrieve with JSONB optimization
async fn get_content_by_title(
    pool: &SqlitePool,
    title: &str,
) -> Result<Vec<ContentMetadata>, sqlx::Error> {
    let rows = sqlx::query!(
        r#"
        SELECT metadata_jsonb
        FROM content_metadata
        WHERE jsonb_extract(metadata_jsonb, '$.title') LIKE ?
        AND expires_at > datetime('now')
        "#,
        format!("%{}%", title)
    )
    .fetch_all(pool)
    .await?;

    let mut results = Vec::new();
    for row in rows {
        if let Some(jsonb_data) = row.metadata_jsonb {
```

```
            // JSONB data is already in binary format, no parsing
needed
            let metadata: ContentMetadata =
serde_json::from_slice(&jsonb_data)?;
            results.push(metadata);
        }
    }

    Ok(results)
}
```

6.2.2.5 Caching Policies

Multi-Tier Caching Strategy:

Cache Tier	Storage Type	TTL Policy	Eviction Strategy
L1 - Memory Cache	HashMap in Rust	5 minutes	LRU with size limits
L2 - SQLite JSONB	Database BLOB	24 hours (search), 7 days (metadata)	TTL-based expiration
L3 - File System	Image files	30 days	LRU with disk space limits

Cache Implementation:

```
-- Cache policy configuration
CREATE TABLE cache_policies (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    cache_type TEXT UNIQUE NOT NULL,
    ttl_seconds INTEGER NOT NULL,
    max_size_mb INTEGER,
    eviction_policy TEXT DEFAULT 'lru',
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Insert default cache policies
INSERT INTO cache_policies (cache_type, ttl_seconds, max_size_mb)
VALUES
```

```
('tmdb_search', 86400, 50),       -- 24 hours, 50MB
('tmdb_metadata', 604800, 200),   -- 7 days, 200MB
('addon_manifest', 3600, 10),     -- 1 hour, 10MB
('user_preferences', 2592000, 5); -- 30 days, 5MB

-- Cache cleanup procedure
CREATE TRIGGER cleanup_expired_cache
AFTER INSERT ON cache_metadata
BEGIN
    DELETE FROM cache_metadata
    WHERE expires_at < datetime('now')
    AND cache_type = NEW.cache_type;
END;
```

6.2.3 COMPLIANCE CONSIDERATIONS

6.2.3.1 Data Retention Rules

Regulatory Compliance Framework:

Data Category	Retention Period	Legal Basis	Deletion Trigger
User Account Data	Account lifetime + 30 days	User consent	Account deletion request
Watch History	2 years or user deletion	Legitimate interest	User request or inactivity
Content Metadata	Cache expiration (7-30 days)	Public data	TTL expiration
Addon Configurations	User-controlled	User consent	Manual removal

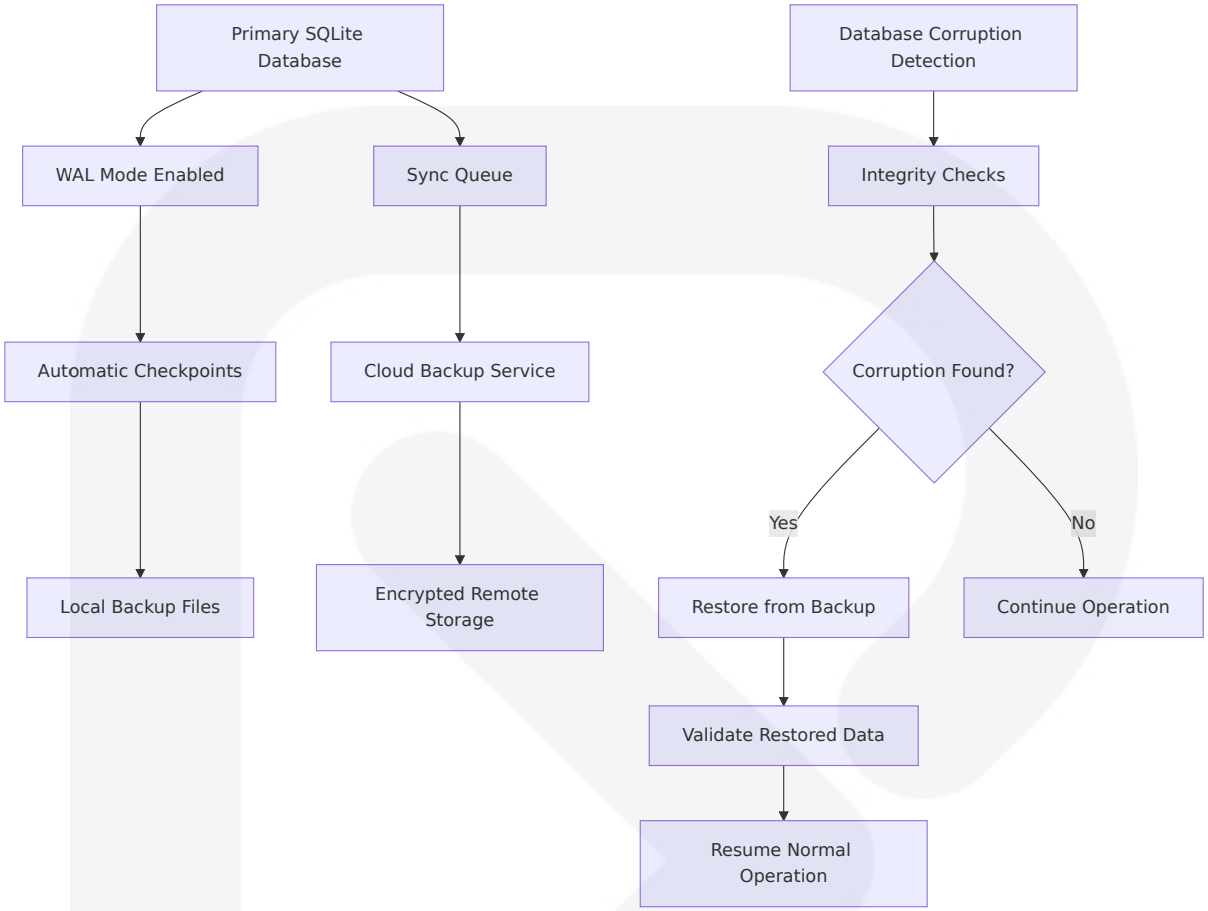
Retention Policy Implementation:

```
-- Data retention policy table
CREATE TABLE data_retention_policies (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    data_category TEXT NOT NULL,
```

```
    retention_days INTEGER NOT NULL,
    legal_basis TEXT NOT NULL,
    auto_delete BOOLEAN DEFAULT TRUE,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Automated retention enforcement
CREATE TRIGGER enforce_user_data_retention
AFTER UPDATE OF last_login ON users
WHEN OLD.last_login < date('now', '-730 days') -- 2 years
BEGIN
    -- Mark user for deletion review
    INSERT INTO deletion_queue (user_id, reason, scheduled_date)
    VALUES (NEW.id, 'inactive_retention', date('now', '+30 days'));
END;

-- GDPR-compliant deletion queue
CREATE TABLE deletion_queue (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id),
    reason TEXT NOT NULL,
    scheduled_date DATE NOT NULL,
    status TEXT DEFAULT 'pending',
    processed_at DATETIME
);
```

6.2.3.2 Backup and Fault Tolerance Policies

Disaster Recovery Architecture:



Fault Tolerance Implementation:

```
-- Backup metadata and validation
CREATE TABLE backup_logs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    backup_type TEXT NOT NULL,
    file_path TEXT,
    file_size INTEGER,
    checksum TEXT NOT NULL,
    validation_status TEXT DEFAULT 'pending',
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    restored_at DATETIME
);

-- Database integrity monitoring
CREATE TABLE integrity_checks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    check_type TEXT NOT NULL,
    result TEXT NOT NULL,
```

```
    error_count INTEGER DEFAULT 0,
    checked_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Automated integrity check trigger
CREATE TRIGGER schedule_integrity_check
AFTER INSERT ON users
WHEN (SELECT COUNT(*) FROM users) % 1000 = 0
BEGIN
    INSERT INTO integrity_checks (check_type, result)
    VALUES ('periodic', 'scheduled');
END;
```

6.2.3.3 Privacy Controls

Privacy-by-Design Implementation:

Privacy Control	Implementation	Technical Measure	User Control
Data Minimization	Store only necessary data	Schema constraints	Opt-in data collection
Purpose Limitation	Use data only for stated purposes	Access control triggers	Purpose consent tracking
Storage Limitation	Automatic data expiration	TTL-based deletion	User-controlled retention
Data Portability	Export functionality	JSON export format	Self-service data export

Privacy Control Schema:

```
-- User privacy preferences
CREATE TABLE privacy_preferences (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER UNIQUE REFERENCES users(id),
    data_collection_consent BOOLEAN DEFAULT FALSE,
    analytics_consent BOOLEAN DEFAULT FALSE,
    sync_consent BOOLEAN DEFAULT FALSE,
    retention_preference_days INTEGER DEFAULT 365,
```

```
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Data processing audit log
CREATE TABLE data_processing_log (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id),
    operation_type TEXT NOT NULL,
    data_category TEXT NOT NULL,
    legal_basis TEXT NOT NULL,
    purpose TEXT NOT NULL,
    processed_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Consent tracking
CREATE TABLE consent_records (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id),
    consent_type TEXT NOT NULL,
    consent_given BOOLEAN NOT NULL,
    consent_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    withdrawal_date DATETIME,
    version TEXT NOT NULL
);
```

6.2.3.4 Audit Mechanisms

Comprehensive Audit Trail:

```
-- Audit log for all database operations
CREATE TABLE audit_log (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    table_name TEXT NOT NULL,
    record_id INTEGER,
    operation TEXT NOT NULL CHECK (operation IN ('INSERT', 'UPDATE',
'DELETE')),
    old_values_jsonb BLOB,
    new_values_jsonb BLOB,
    user_id INTEGER,
    session_id TEXT,
    ip_address TEXT,
```

```
    user_agent TEXT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Security event logging
CREATE TABLE security_events (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    event_type TEXT NOT NULL,
    severity TEXT NOT NULL CHECK (severity IN ('low', 'medium',
'high', 'critical')),
    description TEXT NOT NULL,
    user_id INTEGER,
    ip_address TEXT,
    additional_data_jsonb BLOB,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Audit triggers for sensitive operations
CREATE TRIGGER audit_user_changes
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (
        table_name, record_id, operation,
        old_values_jsonb, new_values_jsonb,
        user_id, timestamp
    ) VALUES (
        'users', NEW.id, 'UPDATE',
        jsonb(json_object('email', OLD.email, 'last_login',
OLD.last_login)),
        jsonb(json_object('email', NEW.email, 'last_login',
NEW.last_login)),
        NEW.id, datetime('now')
    );
END;
```

6.2.3.5 Access Controls

Role-Based Access Control (RBAC):

```sql
-- User roles and permissions
CREATE TABLE user_roles (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER REFERENCES users(id),
    role_name TEXT NOT NULL CHECK (role_name IN ('user', 'admin',
'moderator')),
    granted_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    granted_by INTEGER REFERENCES users(id),
    expires_at DATETIME
);

-- Permission definitions
CREATE TABLE permissions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    permission_name TEXT UNIQUE NOT NULL,
    description TEXT NOT NULL,
    resource_type TEXT NOT NULL,
    action TEXT NOT NULL
);

-- Role-permission mapping
CREATE TABLE role_permissions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    role_name TEXT NOT NULL,
    permission_id INTEGER REFERENCES permissions(id),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Access control enforcement trigger
CREATE TRIGGER enforce_access_control
BEFORE UPDATE ON content_metadata
FOR EACH ROW
WHEN NEW.content_type = 'admin_only'
BEGIN
    SELECT CASE
        WHEN (SELECT COUNT(*) FROM user_roles
              WHERE user_id = NEW.updated_by
              AND role_name IN ('admin', 'moderator')
              AND (expires_at IS NULL OR expires_at >
datetime('now'))) = 0
        THEN RAISE(ABORT, 'Insufficient permissions')
```

```
    END;
END;
```

6.2.4 PERFORMANCE OPTIMIZATION

6.2.4.1 Query Optimization Patterns

JSONB-Optimized Query Patterns:

The function will operate the same in either case, except that it will run faster when the input is JSONB, since it does not need to run the JSON parser. Most SQL functions that return JSON text have a corresponding function that returns the equivalent JSONB.

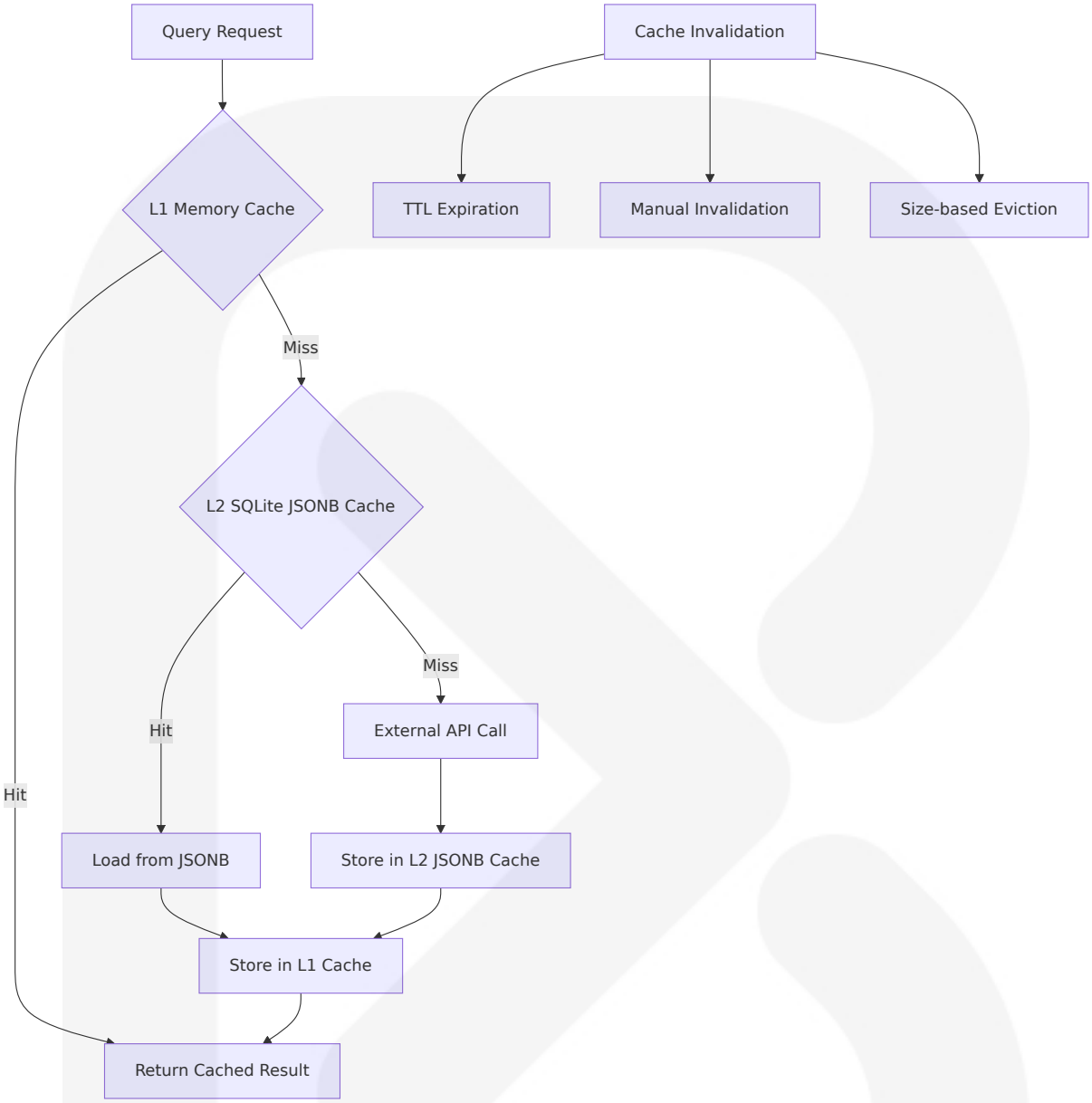
Optimized Query Examples:

```
-- Efficient JSONB queries for content discovery
-- Instead of: json_extract(metadata, '$.title')
-- Use: jsonb_extract(metadata_jsonb, '$.title')

-- Content search with JSONB optimization
SELECT
    id,
    tmdb_id,
    jsonb_extract(metadata_jsonb, '$.title') as title,
    jsonb_extract(metadata_jsonb, '$.vote_average') as rating
FROM content_metadata
WHERE jsonb_extract(metadata_jsonb, '$.title') LIKE ?
  AND jsonb_extract(metadata_jsonb, '$.vote_average') > 7.0
  AND expires_at > datetime('now')
ORDER BY jsonb_extract(metadata_jsonb, '$.popularity') DESC
LIMIT 20;

-- Efficient watch history queries
SELECT
    cm.tmdb_id,
    jsonb_extract(cm.metadata_jsonb, '$.title') as title,
    wh.progress_seconds,
    wh.total_duration_seconds,
```

```
    ROUND((wh.progress_seconds * 100.0 / wh.total_duration_seconds),
2) as progress_percent
FROM watch_history wh
JOIN content_metadata cm ON wh.content_id = cm.id
WHERE wh.user_id = ?
  AND wh.progress_seconds > 0
ORDER BY wh.updated_at DESC
LIMIT 50;

-- Addon manifest queries with JSONB
SELECT
    id,
    url,
    jsonb_extract(manifest_jsonb, '$.name') as name,
    jsonb_extract(manifest_jsonb, '$.version') as version,
    jsonb_extract(manifest_jsonb, '$.types') as supported_types
FROM addons
WHERE status = 'active'
  AND jsonb_extract(manifest_jsonb, '$.types') LIKE '%movie%'
ORDER BY installed_at DESC;
```

6.2.4.2 Caching Strategy

Multi-Level Caching with JSONB:



Cache Implementation with Performance Monitoring:

```
-- Cache performance tracking
CREATE TABLE cache_performance (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    cache_type TEXT NOT NULL,
    hit_count INTEGER DEFAULT 0,
    miss_count INTEGER DEFAULT 0,
    avg_response_time_ms REAL DEFAULT 0,
    last_updated DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Cache optimization view
CREATE VIEW cache_efficiency AS
SELECT
    cache_type,
    hit_count,
    miss_count,
    ROUND((hit_count * 100.0 / (hit_count + miss_count)), 2) as
hit_rate_percent,
    avg_response_time_ms
FROM cache_performance
WHERE hit_count + miss_count > 0;

-- Intelligent cache warming
CREATE TRIGGER warm_popular_content
AFTER INSERT ON watch_history
WHEN (SELECT COUNT(*) FROM watch_history WHERE content_id =
NEW.content_id) > 10
BEGIN
    UPDATE cache_metadata
    SET expires_at = datetime('now', '+14 days')
    WHERE cache_key = 'content_' || NEW.content_id;
END;
```

6.2.4.3 Connection Pooling

SQLite Connection Management with Tauri:

Tauri Plugin providing an interface for the frontend to communicate with SQL databases through sqlx. We use sqlx as the underlying library and adopt their query syntax.

Connection Pool Configuration:

```
use sqlx::{SqlitePool, sqlite::SqlitePoolOptions};
use std::time::Duration;

// Optimized connection pool for desktop application
async fn create_database_pool(database_url: &str) ->
Result<SqlitePool, sqlx::Error> {
    SqlitePoolOptions::new()
```

```
        .max_connections(10) // Suitable for desktop app
        .min_connections(2)  // Keep minimum connections alive
        .acquire_timeout(Duration::from_secs(30))
        .idle_timeout(Duration::from_secs(600)) // 10 minutes
        .max_lifetime(Duration::from_secs(3600)) // 1 hour
        .connect(database_url)
        .await
}

// Connection health monitoring
async fn monitor_connection_health(pool: &SqlitePool) -> Result<(),
sqlx::Error> {
    let health_check = sqlx::query("SELECT 1")
        .fetch_one(pool)
        .await?;

    // Log connection pool statistics
    println!("Active connections: {}", pool.size());
    println!("Idle connections: {}", pool.num_idle());

    Ok(())
}
```

6.2.4.4 Read/Write Splitting

SQLite WAL Mode Optimization:

```
-- Enable WAL mode for better concurrency
PRAGMA journal_mode = WAL;
PRAGMA synchronous = NORMAL;
PRAGMA cache_size = -64000; -- 64MB cache
PRAGMA temp_store = MEMORY;
PRAGMA mmap_size = 268435456; -- 256MB memory mapping

-- Read-optimized queries
CREATE VIEW read_optimized_content AS
SELECT
    id,
    tmdb_id,
    content_type,
    jsonb_extract(metadata_jsonb, '$.title') as title,
```

```
    jsonb_extract(metadata_jsonb, '$.overview') as overview,
    jsonb_extract(metadata_jsonb, '$.poster_path') as poster_path,
    cached_at
FROM content_metadata
WHERE expires_at > datetime('now');

-- Write-optimized batch operations
CREATE TEMPORARY TABLE batch_content_updates (
    tmdb_id INTEGER,
    metadata_json TEXT
);

-- Batch insert procedure
INSERT INTO content_metadata (tmdb_id, content_type, metadata_jsonb,
expires_at)
SELECT
    tmdb_id,
    'movie',
    jsonb(metadata_json),
    datetime('now', '+7 days')
FROM batch_content_updates
ON CONFLICT(tmdb_id) DO UPDATE SET
    metadata_jsonb = jsonb(excluded.metadata_json),
    cached_at = datetime('now');
```

6.2.4.5 Batch Processing Approach

Efficient Batch Operations:

```
// Batch processing for content metadata updates
async fn batch_update_content_metadata(
    pool: &SqlitePool,
    updates: Vec<ContentUpdate>,
) -> Result<(), sqlx::Error> {
    let mut tx = pool.begin().await?;

    // Process in batches of 100 to avoid memory issues
    for chunk in updates.chunks(100) {
        let mut query_builder = sqlx::QueryBuilder::new(
            "INSERT OR REPLACE INTO content_metadata (tmdb_id,
content_type, metadata_jsonb, expires_at) "
```

```
        );

        query_builder.push_values(chunk, |mut b, update| {
            b.push_bind(update.tmdb_id)
             .push_bind(&update.content_type)

.push_bind(serde_json::to_string(&update.metadata).unwrap())
             .push("datetime('now', '+7 days')");
        });

        query_builder.build().execute(&mut *tx).await?;
    }

    tx.commit().await?;
    Ok(())
}

// Batch cleanup operations
async fn batch_cleanup_expired_data(pool: &SqlitePool) -> Result<u64,
sqlx::Error> {
    let mut tx = pool.begin().await?;

    // Clean expired cache entries
    let cache_deleted = sqlx::query!(
        "DELETE FROM cache_metadata WHERE expires_at <
datetime('now')"
    )
    .execute(&mut *tx)
    .await?
    .rows_affected();

    // Clean expired content metadata
    let content_deleted = sqlx::query!(
        "DELETE FROM content_metadata WHERE expires_at <
datetime('now')"
    )
    .execute(&mut *tx)
    .await?
    .rows_affected();

    // Update statistics
    sqlx::query!(
        r#"
```

```
        INSERT INTO cleanup_stats (cache_deleted, content_deleted,
cleanup_date)
        VALUES (?, ?, datetime('now'))
        "#,
        cache_deleted,
        content_deleted
    )
    .execute(&mut *tx)
    .await?;

    tx.commit().await?;
    Ok(cache_deleted + content_deleted)
}
```

Performance Monitoring and Optimization:

```
-- Query performance tracking
CREATE TABLE query_performance (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    query_type TEXT NOT NULL,
    execution_time_ms REAL NOT NULL,
    rows_affected INTEGER,
    cache_hit BOOLEAN DEFAULT FALSE,
    executed_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Performance optimization recommendations
CREATE VIEW performance_recommendations AS
SELECT
    query_type,
    COUNT(*) as execution_count,
    AVG(execution_time_ms) as avg_time_ms,
    MAX(execution_time_ms) as max_time_ms,
    SUM(CASE WHEN cache_hit THEN 1 ELSE 0 END) * 100.0 / COUNT(*) as
cache_hit_rate
FROM query_performance
WHERE executed_at > datetime('now', '-7 days')
GROUP BY query_type
HAVING execution_count > 10
ORDER BY avg_time_ms DESC;
```

This comprehensive database design leverages SQLite's JSONB capabilities for optimal performance while maintaining data integrity, security, and compliance requirements for the Stremio Clone Desktop Application. The design supports efficient content discovery, user management, addon system functionality, and provides robust backup and recovery mechanisms suitable for a desktop media center application.

6.3 Integration Architecture

6.3.1 API DESIGN

6.3.1.1 Protocol Specifications

The Stremio Clone Desktop Application implements a multi-protocol integration architecture leveraging Tauri's particular style of Inter-Process Communication called Asynchronous Message Passing, where processes exchange requests and responses serialized using some simple data representation.

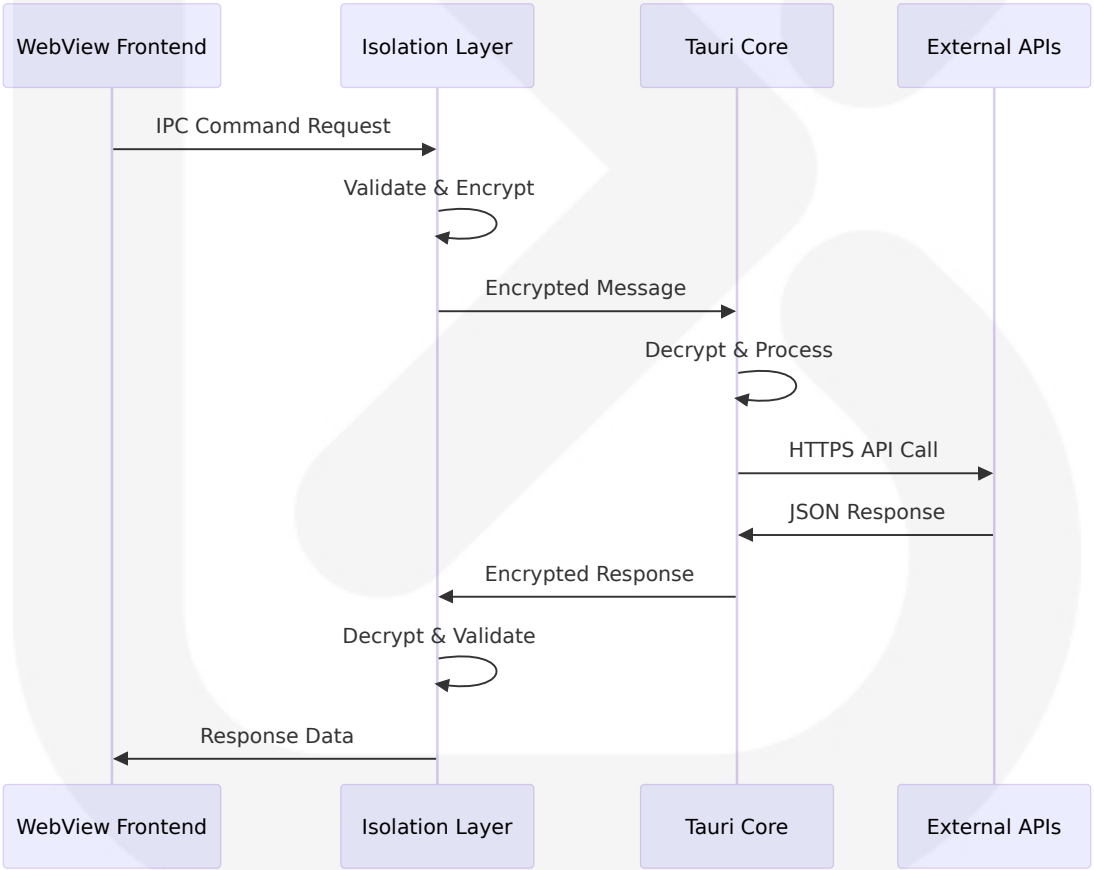
Core Protocol Stack:

Protocol Layer	Implementation	Purpose	Security Features
Tauri IPC	JSON-RPC over secure channels	Frontend-Backend communication	Cryptographically secure key generated once each time the Tauri application is started, encrypted using the browser's SubtleCrypto implementation
HTTPS/REST	HTTP/1.1 and HTTP/2	External API communication	TLS 1.3, certificate validation
WebSocket	Secure WebSocket (WSS)	Real-time addon communication	Connection-level encryption

Protocol Layer	Implementation	Purpose	Security Features
Custom Protocol	Stremio Addon Protocol	Addon manifest and stream delivery	CORS enforcement, HTTPS requirement

IPC Protocol Specification:

The primary API, invoke, is similar to the browser's fetch API and allows the Frontend to invoke Rust functions, pass arguments, and receive data. Because this mechanism uses a JSON-RPC like protocol under the hood to serialize requests and responses, all arguments and return data must be serializable to JSON.



6.3.1.2 Authentication Methods

Multi-Tier Authentication Architecture:

Authentication Tier	Method	Implementation	Token Management
Local Authentication	Bcrypt + Session Tokens	SQLite credential storage	30-day expiration
TMDB API	API Key Authentication	Before being issued an API key you will have to agree to our terms of use	Static API key
Addon Authentication	URL-based validation	HTTPS enforcement	No persistent tokens
IPC Authentication	Capability-based access	Controls application windows' and webviews' fine grained access to the Tauri core, application, or plugin commands. If a webview or its window is not matching any capability then it has no access to the IPC layer at all	Runtime capabilities

Authentication Flow Implementation:

```
// Tauri command with authentication validation
#[tauri::command]
async fn authenticated_search(
    query: String,
    state: tauri::State<'_, AppState>,
    window: tauri::Window,
) -> Result<SearchResults, AuthError> {
    // Validate window capability
    if !window.has_capability("content:search") {
        return Err(AuthError::InsufficientPermissions);
    }

    // Validate user session
    let user_session =
state.auth_manager.validate_session(&window).await?;
```

```
    // Proceed with authenticated operation
    state.content_service.search(&query, &user_session).await
}
```

6.3.1.3 Authorization Framework

Capability-Based Authorization:

Tauri provides application and plugin developers with a capabilities system, to granually enable and constrain the core exposure to the application frontend running in the system WebView. Capabilities define which permissions are granted or denied for which windows or webviews.

Authorization Matrix:

Resource Type	Guest Mode	Authenticated User	Admin User	Required Capability
Content Search	Read-only	Full access	Full access	`content:search`
Library Management	Disabled	Full access	Full access	`library:manage`
Addon Installation	Disabled	User addons only	All addons	`addon:install`
System Settings	Disabled	User settings	Global settings	`system:configure`

Capability Configuration:

```
{
  "identifier": "content-access",
  "description": "Allows content discovery and search operations",
  "windows": ["main-window"],
  "permissions": [
    "content:search",
    "content:metadata",
    "tmdb:api-access"
  ],
```

```
  "platforms": ["linux", "macOS", "windows"]
}
```

6.3.1.4 Rate Limiting Strategy

TMDB API Rate Limiting:

While our legacy rate limits have been disabled for some time, we do still have some upper limits to help mitigate needlessly high bulk scraping. They sit somewhere in the 50 requests per second range. Additionally, I believe it's a maximum of: 50 requests per second and 20 connections per IP.

Rate Limiting Implementation:

Service	Rate Limit	Connection Limit	Strategy
TMDB API	50 requests/second	20 connections/IP	Token bucket algorithm
Addon Requests	10 requests/second per addon	5 connections/addon	Sliding window
IPC Commands	100 commands/second	N/A	Leaky bucket
Image Downloads	For image.tmdb.org the only thing we limit is the max number of simultaneous connections. The limit is the same, 20	20 connections	Connection pooling

Rate Limiter Implementation:

```
use std::time::{Duration, Instant};
use tokio::sync::Semaphore;

pub struct RateLimiter {
    semaphore: Semaphore,
```

```
    last_request: tokio::sync::Mutex<Instant>,
    min_interval: Duration,
}

impl RateLimiter {
    pub fn new(max_concurrent: usize, requests_per_second: u32) ->
Self {
        Self {
            semaphore: Semaphore::new(max_concurrent),
            last_request: tokio::sync::Mutex::new(Instant::now()),
            min_interval: Duration::from_millis(1000 /
requests_per_second as u64),
        }
    }

    pub async fn acquire(&self) -> Result<(), RateLimitError> {
        let _permit = self.semaphore.acquire().await?;

        let mut last = self.last_request.lock().await;
        let now = Instant::now();
        let elapsed = now.duration_since(*last);

        if elapsed < self.min_interval {
            tokio::time::sleep(self.min_interval - elapsed).await;
        }

        *last = Instant::now();
        Ok(())
    }
}
```

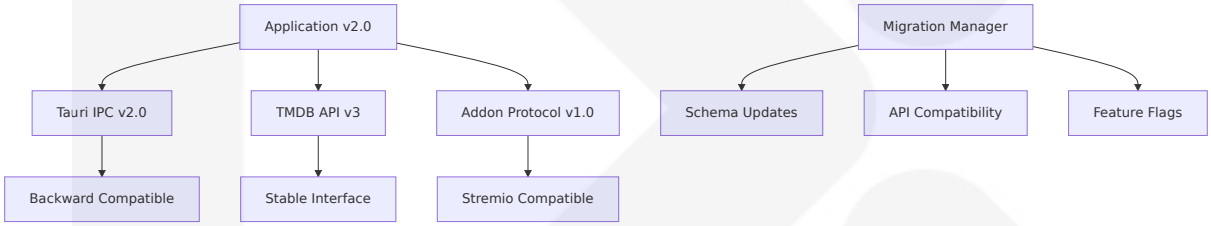
6.3.1.5 Versioning Approach

API Versioning Strategy:

Component	Versioning Scheme	Compatibility	Migration Path
Tauri IPC	Semantic versioning	Backward compatible	Automatic migration

Component	Versioning Scheme	Compatibility	Migration Path
TMDB API	Get started with the basics of the TMDB API v3	Stable API	Version pinning
Addon Protocol	Stremio-compatible	The first thing to define for your addon is the manifest, which describes it's name, purpose and some technical details	Manifest versioning
Database Schema	Migration-based	Forward compatible	Incremental migrations

Version Compatibility Matrix:



6.3.1.6 Documentation Standards

API Documentation Framework:

Documentation Type	Format	Generation	Maintenance
Tauri Commands	Rust doc comments	Automatic via rustdoc	Inline documentation
IPC Interface	TypeScript definitions	Generated from Rust types	Type-safe generation
External APIs	OpenAPI/Swagger	Manual documentation	Version-controlled
Addon Protocol	Markdown specifications	Manual maintenance	Community contributions

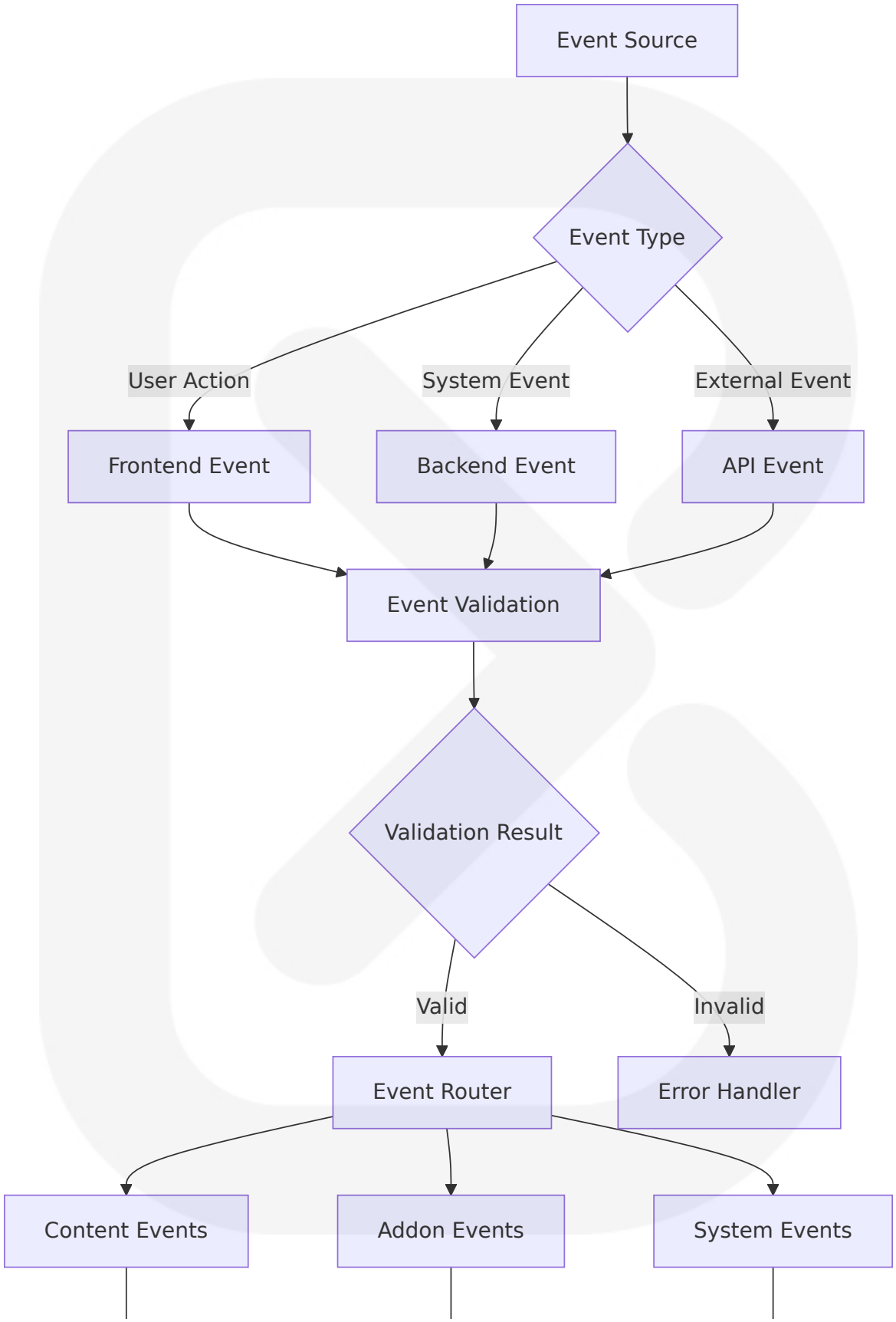
6.3.2 MESSAGE PROCESSING

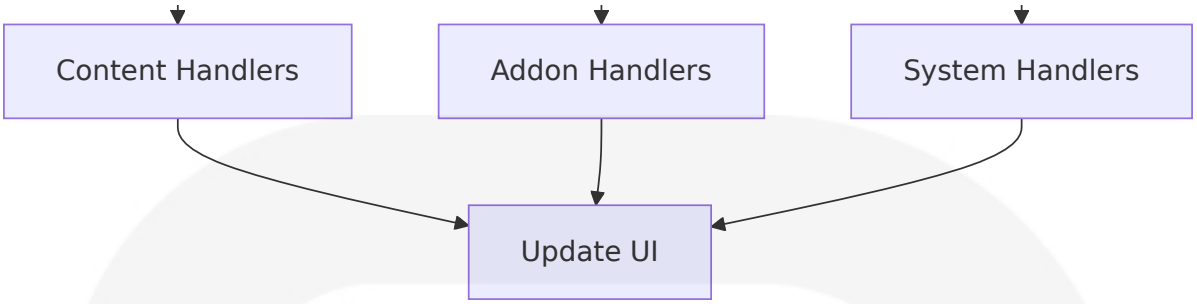
6.3.2.1 Event Processing Patterns

Event-Driven Architecture:

Unlike Commands, Events can be emitted by both the Frontend and the Tauri Core. Events sent between the Core and the Webview.

Event Processing Flow:


Event Source
Event Type
User Action
System Event
External Event
Frontend Event
Backend Event
API Event
Event Validation
Validation Result
Valid
Invalid
Event Router
Error Handler
Content Events
Addon Events
System Events



Event Categories:

Event Category	Source	Processing Pattern	Persistence
Content Discovery	User search, API responses	Async processing with caching	Temporary cache
Addon Management	User actions, health checks	State machine pattern	Configuration storage
Playback Events	Media player, user controls	Real-time processing	Progress tracking
System Events	OS notifications, network changes	Event aggregation	System logs

6.3.2.2 Message Queue Architecture

Async Message Processing:

The application implements an in-memory message queue for handling asynchronous operations and background tasks.

Queue Architecture:

```
use tokio::sync::mpsc;
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum MessageType {
    ContentSearch { query: String, user_id: Option<u64> },
    AddonHealthCheck { addon_id: u64 },
    MetadataUpdate { content_id: u64 },
    UserSync { user_id: u64 },
```

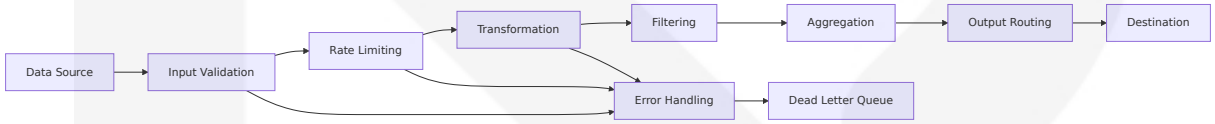
```
}

pub struct MessageQueue {
    sender: mpsc::UnboundedSender<MessageType>,
    receiver:
tokio::sync::Mutex<mpsc::UnboundedReceiver<MessageType>>,
}

impl MessageQueue {
    pub fn new() -> Self {
        let (sender, receiver) = mpsc::unbounded_channel();
        Self {
            sender,
            receiver: tokio::sync::Mutex::new(receiver),
        }
    }

    pub async fn process_messages(&self, handlers: &MessageHandlers) {
        let mut receiver = self.receiver.lock().await;
        while let Some(message) = receiver.recv().await {
            match message {
                MessageType::ContentSearch { query, user_id } => {
                    handlers.handle_content_search(query,
user_id).await;
                }
                MessageType::AddonHealthCheck { addon_id } => {

handlers.handle_addon_health_check(addon_id).await;
                }
                // Handle other message types...
            }
        }
    }
}
```

6.3.2.3 Stream Processing Design

Real-Time Data Streams:

Stream Type	Source	Processing	Destination
Search Results	TMDB API	Filtering, ranking	Frontend display
Addon Responses	Remote addons	Validation, caching	Stream aggregation
Playback Progress	Media player	Throttling, persistence	User database
Health Monitoring	System metrics	Aggregation, alerting	Monitoring dashboard

Stream Processing Pipeline:



6.3.2.4 Batch Processing Flows

Background Batch Operations:

Batch Operation	Frequency	Batch Size	Processing Time
Metadata Refresh	Every 6 hours	100 items	5-10 minutes
Addon Health Checks	Every 30 minutes	All active addons	2-5 minutes
Cache Cleanup	Daily	All expired entries	1-2 minutes
User Data Sync	Every 15 minutes	Per-user changes	30 seconds

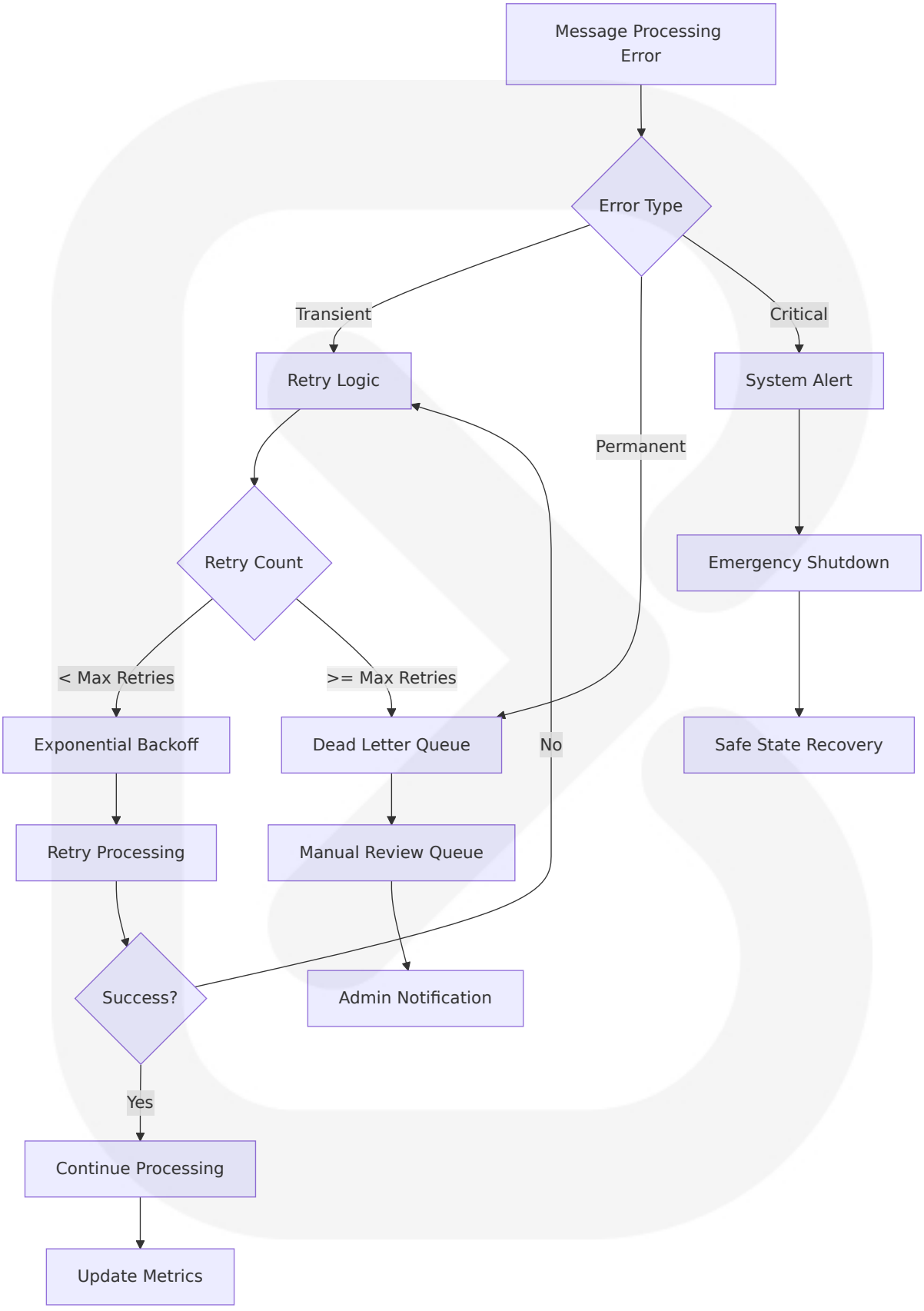
Batch Processing Implementation:

```
pub struct BatchProcessor {
    batch_size: usize,
    processing_interval: Duration,
```

```rust
}

impl BatchProcessor {
    pub async fn process_metadata_updates(&self, pool: &SqlitePool) ->
Result<(), ProcessingError> {
        let expired_content = sqlx::query!(
            "SELECT id, tmdb_id FROM content_metadata
             WHERE expires_at < datetime('now')
             LIMIT ?",
            self.batch_size as i64
        )
        .fetch_all(pool)
        .await?;

        for chunk in expired_content.chunks(10) {
            let futures = chunk.iter().map(|content| {
                self.refresh_content_metadata(content.tmdb_id)
            });

            try_join_all(futures).await?;

            // Rate limiting between batches
            tokio::time::sleep(Duration::from_millis(100)).await;
        }

        Ok(())
    }
}
```

6.3.2.5 Error Handling Strategy

Comprehensive Error Recovery:


Message Processing Error
Error Type
Transient
Critical
Permanent
Retry Logic
System Alert
Retry Count
Emergency Shutdown
< Max Retries
>= Max Retries
Exponential Backoff
Dead Letter Queue
Safe State Recovery
Retry Processing
Manual Review Queue
Success?
No
Admin Notification
Yes
Continue Processing
Update Metrics

Error Classification:

Error Type	Handling Strategy	Recovery Action	Notification Level
Network Timeout	Exponential backoff retry	Automatic retry up to 3 times	Debug log
API Rate Limit	Delay and retry	Wait for rate limit reset	Info log
Invalid Data	Skip and continue	Log error, continue processing	Warning log
System Resource	Graceful degradation	Reduce processing load	Error alert

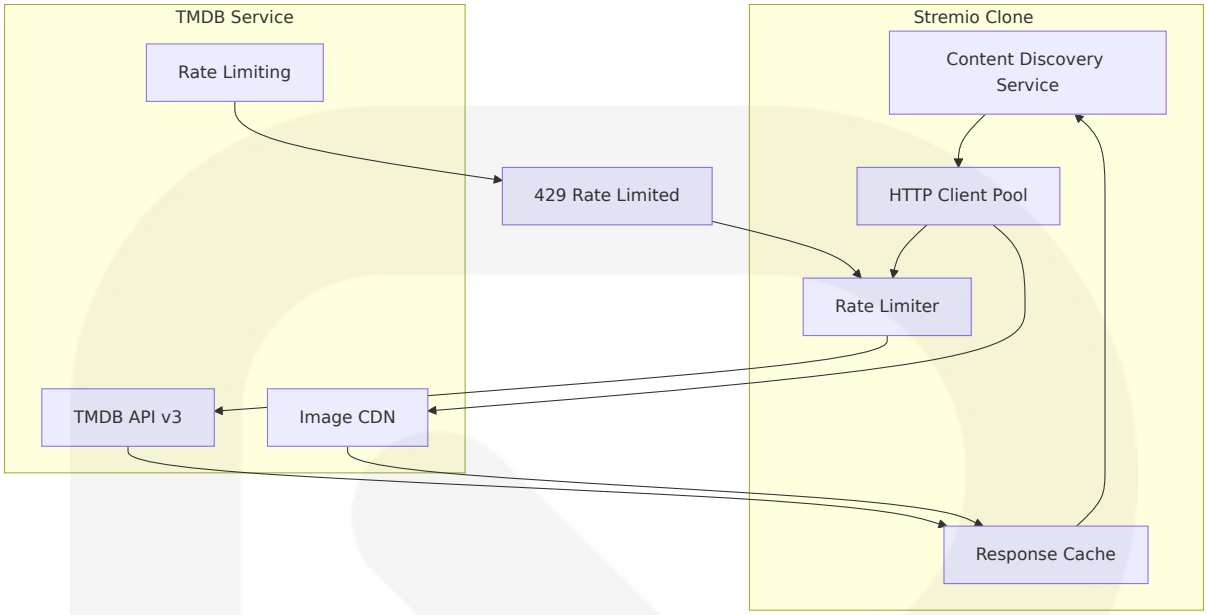
6.3.3 EXTERNAL SYSTEMS

6.3.3.1 Third-Party Integration Patterns

TMDB API Integration:

Our API is free to use for non-commercial purposes as long as you attribute TMDB as the source of the data and/or images. We do not currently provide an SLA. However, we do make every reasonable attempt to keep our service online and accessible.

Integration Architecture:



TMDB Integration Specifications:

Integration Aspect	Specification	Implementation
Attribution	"This product uses the TMDB API but is not endorsed or certified by TMDB."	Displayed in About section
Rate Limits	Our rate limits are really high (~40 r/s per IP address), so this rarely causes people an issue	Token bucket implementation
SSL/TLS	We strongly recommend you use SSL	HTTPS enforcement

6.3.3.2 Legacy System Interfaces

Stremio Addon Compatibility:

The application maintains compatibility with the existing Stremio addon ecosystem through protocol adherence.

Addon Protocol Specifications:

Protocol Element	Requirement	Validation
Manifest Format	The skeleton of a Stremio add-on's manifest JSON structure	Schema validation

| **HTTPS Requirement** | Every add-on must provide CORS headers for its resources. Stremio cannot make use of an add-on that does not support CORS | URL protocol validation |
| **Resource Endpoints** | Every resource is accessed at a certain endpoint where your add-on should respond with proper data | Endpoint availability checks |

Addon Manifest Validation:

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Serialize, Deserialize)]
pub struct AddonManifest {
    pub id: String,
    pub version: String,
    pub name: String,
    pub description: String,
    pub logo: Option<String>,
    pub resources: Vec<String>,
    pub types: Vec<String>,
    pub catalogs: Option<Vec<Catalog>>,
}

pub async fn validate_addon_manifest(url: &str) ->
Result<AddonManifest, ValidationError> {
    // Ensure HTTPS (except localhost)
    if !url.starts_with("https://") &&
!url.starts_with("http://127.0.0.1") {
        return Err(ValidationError::InsecureProtocol);
    }

    let response = reqwest::get(&format!("{}/manifest.json",
url)).await?;

    // Validate CORS headers
    if !response.headers().contains_key("access-control-allow-origin")
{
        return Err(ValidationError::MissingCors);
    }

    let manifest: AddonManifest = response.json().await?;
```

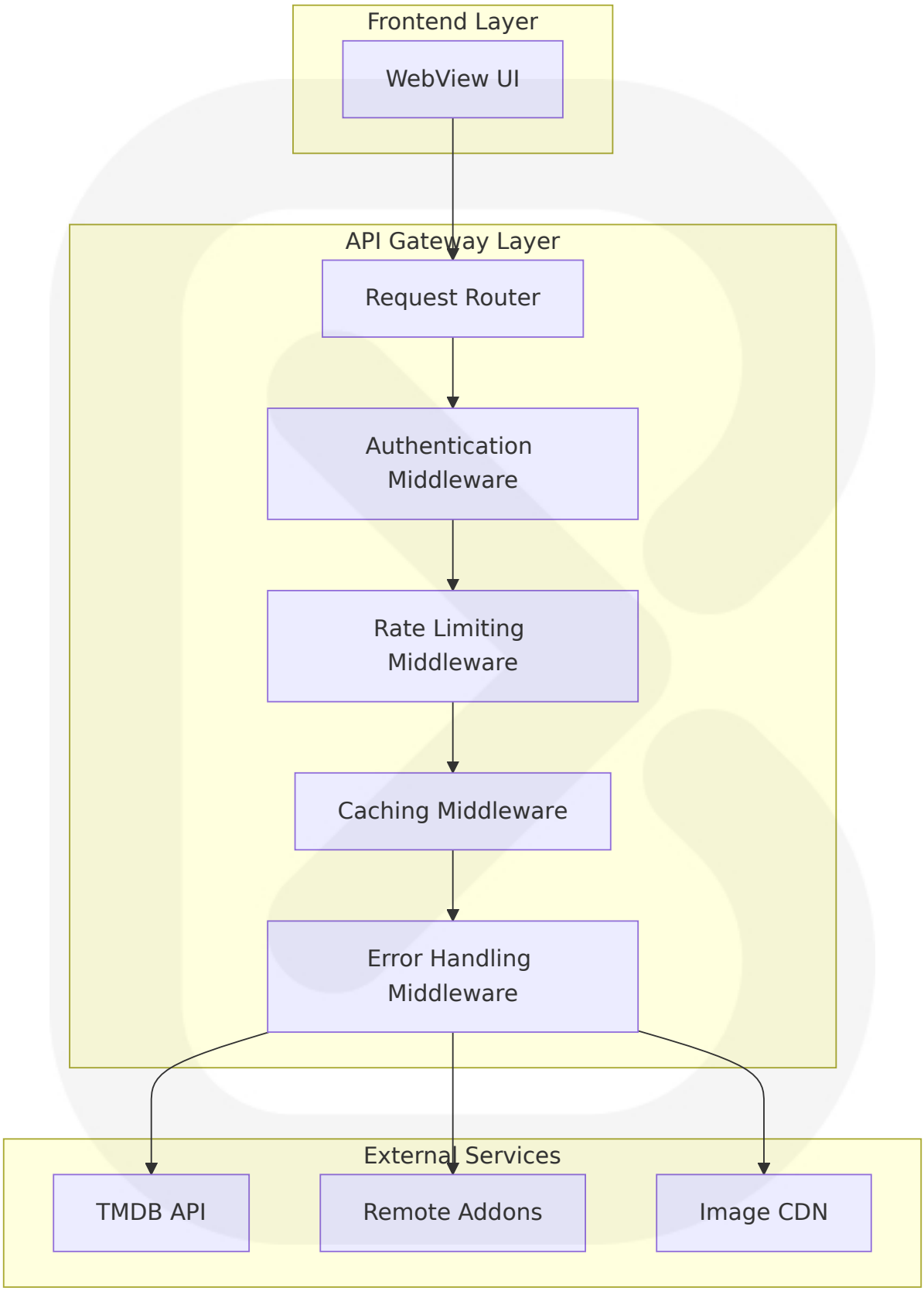
```
    // Validate required fields
    if manifest.id.is_empty() || manifest.name.is_empty() {
        return Err(ValidationError::MissingRequiredFields);
    }

    Ok(manifest)
}
```

6.3.3.3 API Gateway Configuration

Internal API Gateway Pattern:

The application implements an internal API gateway pattern to manage external service interactions.

Gateway Architecture:


Frontend Layer
WebView UI
API Gateway Layer
Request Router
Authentication Middleware
Rate Limiting Middleware
Caching Middleware
Error Handling Middleware
External Services
TMDB API
Remote Addons
Image CDN

Gateway Configuration:

Service Route	Middleware Stack	Caching Policy	Error Handling
`/api/tmdb/*`	Auth, RateLimit, Cache	24h for metadata, 1h for search	Retry with backoff
`/api/addon/*`	Validation, RateLimit	1h for manifests, 5m for streams	Circuit breaker
`/api/images/*`	Cache, Compression	30 days	Fallback to placeholder

6.3.3.4 External Service Contracts

Service Level Agreements:

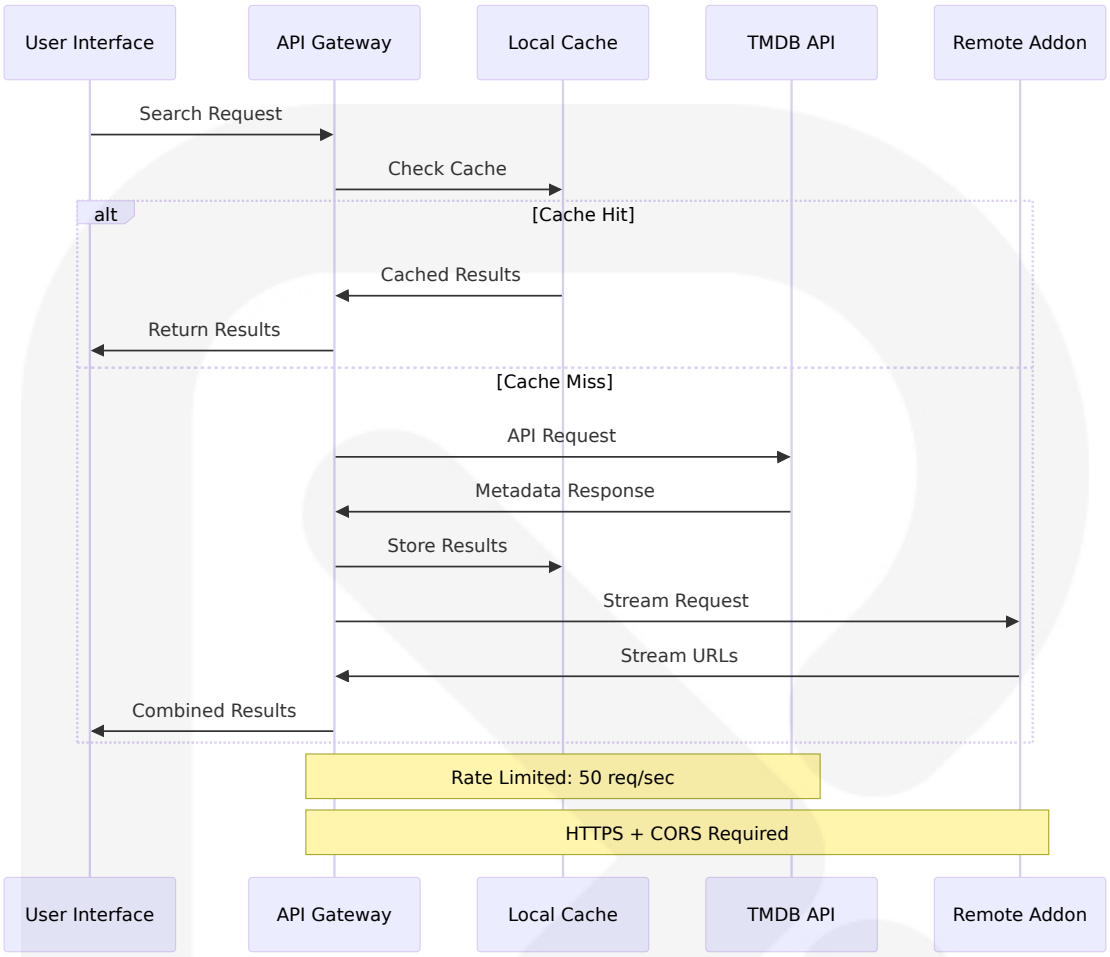
Service	Availability	Response Time	Error Rate	Fallback Strategy
TMDB API	We do not currently provide an SLA. However, we do make every reasonable attempt to keep our service online and accessible	< 3 seconds	< 5%	Local cache fallback
Remote Addons	Variable	< 10 seconds	< 10%	Disable failed addons
Image CDN	99.9%	< 2 seconds	< 1%	Placeholder images

Contract Monitoring:

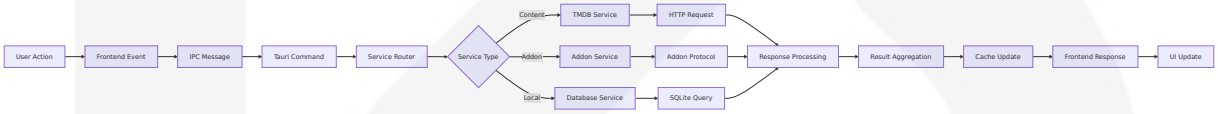
```
pub struct ServiceMonitor {
    metrics: Arc<Mutex<ServiceMetrics>>,
}

impl ServiceMonitor {
    pub async fn check_service_health(&self, service: &str) ->
```

```
ServiceHealth {
        let start = Instant::now();

        match self.ping_service(service).await {
            Ok(_) => {
                let response_time = start.elapsed();
                self.record_success(service, response_time).await;

                if response_time > Duration::from_secs(5) {
                    ServiceHealth::Degraded
                } else {
                    ServiceHealth::Healthy
                }
            }
            Err(e) => {
                self.record_failure(service, &e).await;
                ServiceHealth::Unhealthy
            }
        }
    }
}
```

Integration Flow Diagrams:



Message Flow Architecture:



This comprehensive integration architecture ensures secure, performant, and reliable communication between the Stremio Clone Desktop Application and all external services while maintaining compatibility with existing Stremio addon ecosystem and providing robust error handling and monitoring capabilities.

6.4 Security Architecture

6.4.1 AUTHENTICATION FRAMEWORK

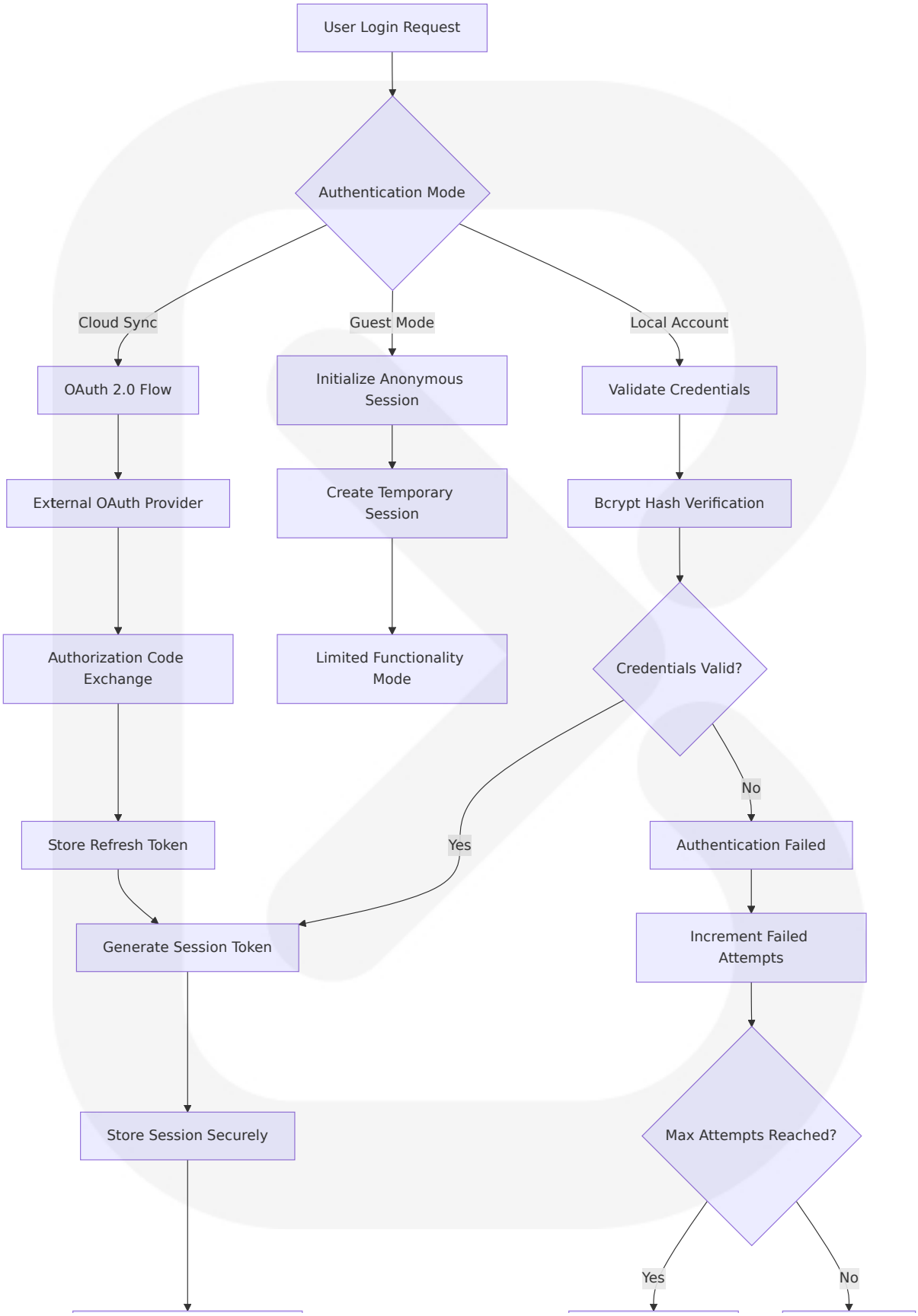
6.4.1.1 Identity Management

The Stremio Clone Desktop Application implements a multi-tier identity management system leveraging Tauri's security model that differentiates between Rust code written for the application's core and frontend code written in any framework or language understood by the system WebView.

Identity Architecture:

Identity Tier	Implementation	Security Features	Storage Method
Local Identity	SQLite-based user accounts	Bcrypt password hashing, session tokens	Encrypted local database
Guest Mode	Anonymous sessions	No persistent data, limited functionality	Memory-only storage
Optional Cloud Sync	OAuth 2.0 integration	Refresh tokens, secure API communication	Encrypted cloud storage

Identity Validation Process:


User Login Request
Authentication Mode
Cloud Sync
Guest Mode
Local Account
OAuth 2.0 Flow
Initialize Anonymous Session
Validate Credentials
External OAuth Provider
Create Temporary Session
Bcrypt Hash Verification
Authorization Code Exchange
Limited Functionality Mode
Credentials Valid?
No
Store Refresh Token
Yes
Authentication Failed
Generate Session Token
Increment Failed Attempts
Store Session Securely
Max Attempts Reached?
Yes
No

6.4.1.2 Multi-Factor Authentication

MFA Implementation Strategy:

The application supports optional multi-factor authentication for enhanced security, particularly for cloud synchronization features.

MFA Method	Implementation	Use Case	Security Level
TOTP (Time-based OTP)	Standard authenticator apps	Cloud sync authentication	High
Hardware Keys	FIDO2/WebAuthn support	High-security environments	Very High
Backup Codes	One-time recovery codes	Account recovery	Medium

6.4.1.3 Session Management

Session Security Architecture:

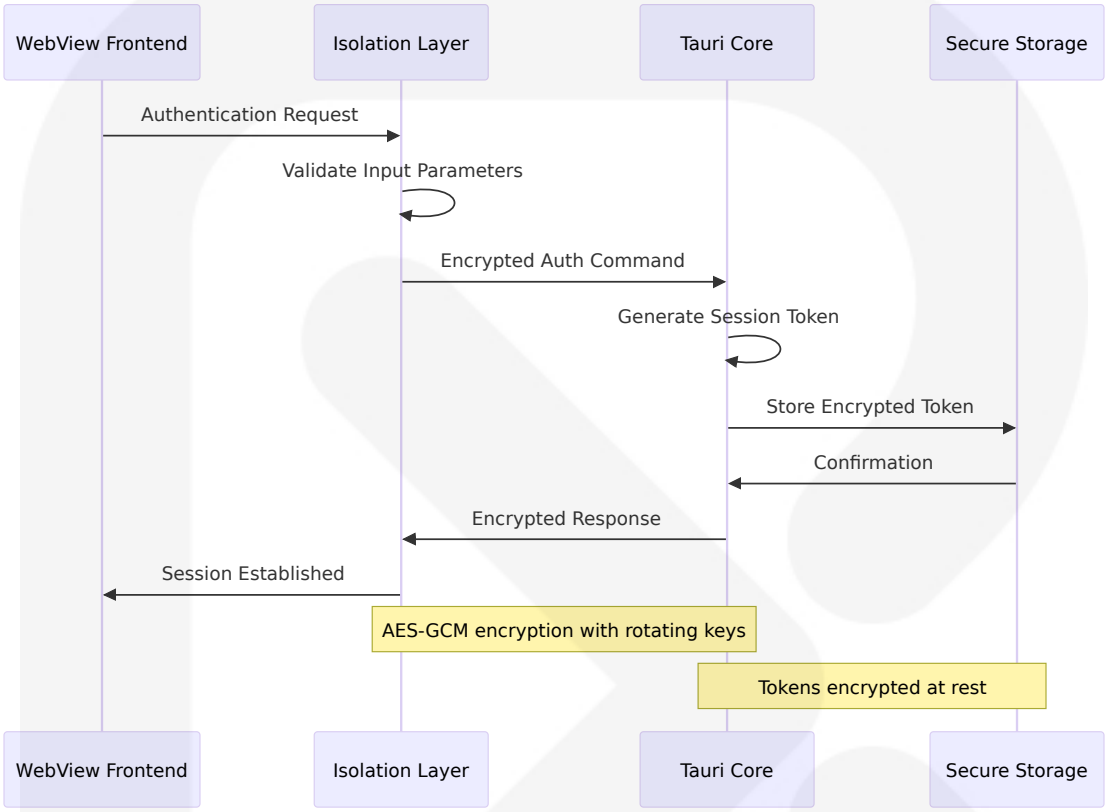
There is a cryptographically secure key generated once each time the Tauri application is started. It is not generally noticeable if the system already has enough entropy to immediately return enough random numbers, which is extremely common for desktop environments.

Session Management Matrix:

Session Type	Duration	Storage	Validation Method
Local Sessions	30 days	Encrypted SQLite	JWT with rotating secrets
Guest Sessions	Application lifetime	Memory only	Temporary tokens
Cloud Sessions	7 days	Secure keychain	OAuth refresh tokens

6.4.1.4 Token Handling

Secure Token Architecture:



Token Security Specifications:

Token Type	Algorithm	Expiration	Rotation Policy
Session Tokens	JWT with RS256	24 hours	Daily rotation
Refresh Tokens	Secure random 256-bit	30 days	On use rotation
API Tokens	Bearer tokens	1 hour	Automatic refresh

6.4.1.5 Password Policies

Password Security Requirements:

Policy Category	Requirement	Implementation	Validation
Complexity	Minimum 12 characters, mixed case, numbers, symbols	Client-side validation with server verification	Real-time strength meter
History	Cannot reuse last 5 passwords	Bcrypt hash comparison	Database history check
Expiration	Optional 90-day expiration	Configurable policy	Automated notifications
Breach Detection	Check against known breaches	HaveIBeenPwned API integration	Registration/change validation

6.4.2 AUTHORIZATION SYSTEM

6.4.2.1 Role-Based Access Control

Tauri Capabilities System:

Tauri provides application and plugin developers with a capabilities system, to granually enable and constrain the core exposure to the application frontend running in the system WebView. Capabilities define which permissions are granted or denied for which windows or webviews.

RBAC Implementation:

Role	Capabilities	System Access	Addon Permissions
Guest User	`content:search`, `ui:basic`	Read-only content discovery	No addon installation
Authenticated User	`content:manage`, `library:full`, `addon:install`	Full library management	User addon installation
Admin User	`system:configure`, `addon:manage`	System settings	Global addon management

Capability Configuration Example:

```
{
  "identifier": "authenticated-user-capability",
  "description": "Standard user permissions for content and library
management",
  "windows": ["main-window"],
  "permissions": [
    "content:search",
    "content:metadata",
    "library:manage",
    "addon:install-user",
    "sync:user-data"
  ],
  "platforms": ["linux", "macOS", "windows"]
}
```

6.4.2.2 Permission Management

Granular Permission System:

A grouping and boundary mechanism developers can use to isolate access to the IPC layer. It controls application windows' and webviews' fine grained access to the Tauri core, application, or plugin commands. If a webview or its window is not matching any capability then it has no access to the IPC layer at all.

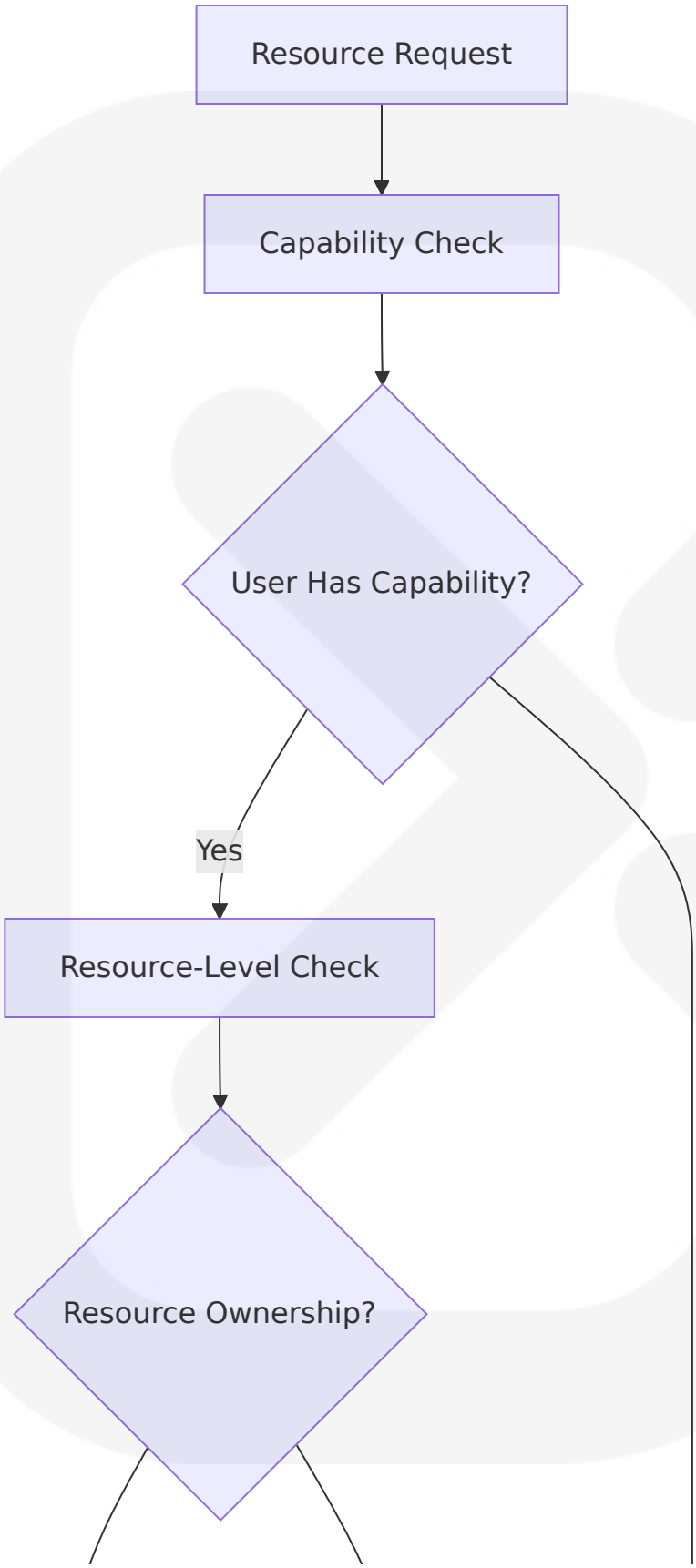
Permission Matrix:

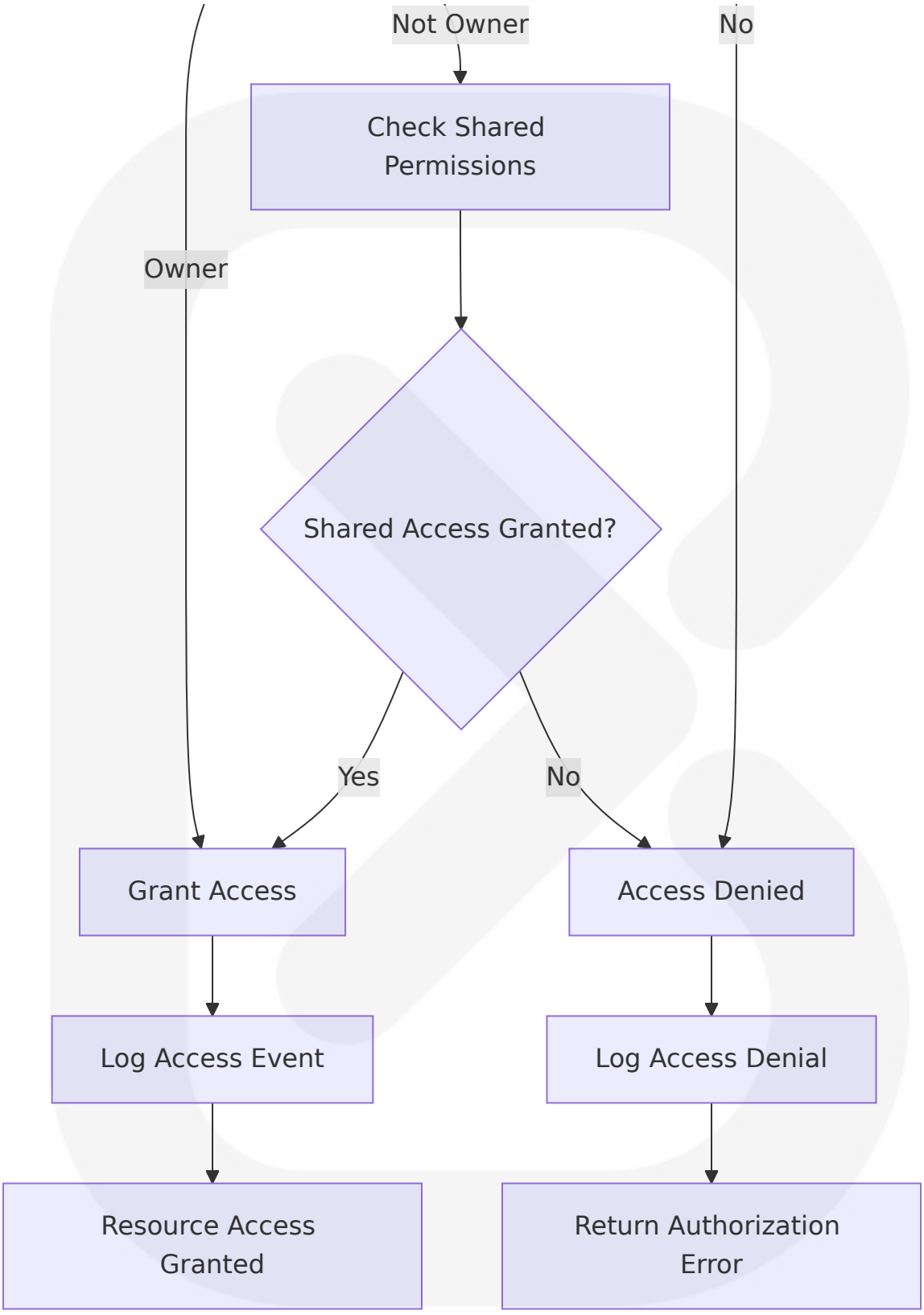
Permission Category	Guest Mode	Authenticated User	Admin User
Content Discovery	□ Read-only	□ Full access	□ Full access
Library Management	□ Disabled	□ Personal library	□ All libraries
Addon Installation	□ Disabled	□ User addons	□ System addons

Permission Category	Guest Mode	Authenticated User	Admin User
System Configuration	□ Disabled	□ Disabled	□ Full access

6.4.2.3 Resource Authorization

Resource Access Control:


Resource Request
Capability Check
User Has Capability?
Yes
Resource-Level Check
Resource Ownership?



6.4.2.4 Policy Enforcement Points

Security Enforcement Architecture:

Enforcement Point	Location	Validation Method	Fallback Action
IPC Layer	Tauri command handlers	Capability validation	Command rejection
Database Layer	SQL query interceptors	User context validation	Query blocking
File System	Tauri FS plugin	Path validation	Access denial
Network Requests	HTTP client middleware	URL whitelist validation	Request blocking

6.4.2.5 Audit Logging

Comprehensive Audit Trail:

```
-- Security audit log table
CREATE TABLE security_audit_log (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER,
    session_id TEXT NOT NULL,
    action_type TEXT NOT NULL,
    resource_type TEXT NOT NULL,
    resource_id TEXT,
    permission_required TEXT NOT NULL,
    access_granted BOOLEAN NOT NULL,
    ip_address TEXT,
    user_agent TEXT,
    additional_context TEXT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Audit trigger for sensitive operations
CREATE TRIGGER audit_addon_installation
AFTER INSERT ON user_addons
FOR EACH ROW
BEGIN
    INSERT INTO security_audit_log (
        user_id, action_type, resource_type, resource_id,
```

```
        permission_required, access_granted, timestamp
    ) VALUES (
        NEW.user_id, 'ADDON_INSTALL', 'addon', NEW.addon_id,
        'addon:install-user', TRUE, datetime('now')
    );
END;
```

6.4.3 DATA PROTECTION

6.4.3.1 Encryption Standards

Multi-Layer Encryption Strategy:

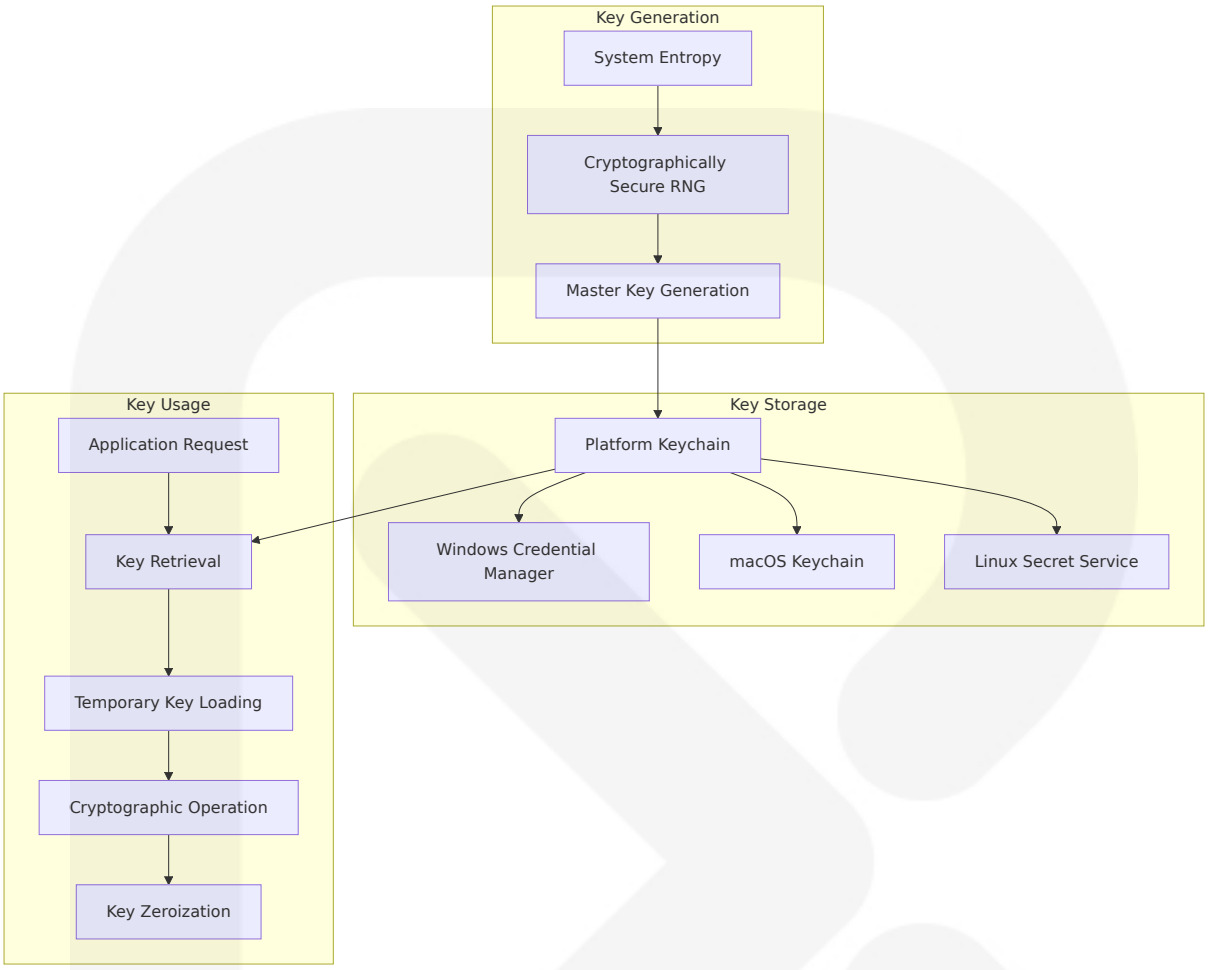
Data encryption at rest is a mandatory step toward data privacy, compliance, and data sovereignty. Best practice: Apply disk encryption to help safeguard your data.

Encryption Implementation Matrix:

Data Category	Encryption Method	Key Management	Performance Impact
User Credentials	Bcrypt with salt (cost factor 12)	Application-managed	Minimal
Session Tokens	AES-256-GCM	Hardware-backed keys	Negligible
Local Database	SQLite with SQLCipher	User-derived keys	Low
IPC Communication	AES-GCM (the only authenticated mode algorithm included in SubtleCrypto)	Runtime-generated keys	Low

6.4.3.2 Key Management

Secure Key Architecture:



Key Management Policies:

Key Type	Generation	Storage	Rotation	Backup
Master Keys	Hardware RNG	Platform keychain	Annual	Secure export
Session Keys	Cryptographically secure key generated once each time the Tauri application is started	Memory only	Per session	Not backed up
Database Keys	PBKDF2 from user password	Encrypted storage	On password change	User responsibility

6.4.3.3 Data Masking Rules

Sensitive Data Protection:

Data Type	Masking Strategy	Display Format	Storage Format
Email Addresses	Partial masking	`u***@example.com`	Full encryption
API Keys	Prefix only	`sk_live_****`	Full encryption
User Passwords	Never displayed	`••••••••`	Bcrypt hash
Session Tokens	Never logged	`[REDACTED]`	Encrypted storage

6.4.3.4 Secure Communication

Network Security Architecture:

Please note: addon URLs in Stremio must be loaded with HTTPS (except 127.0.0.1) and must support CORS! CORS support is handled automatically by the SDK, but if you're trying to load your addon remotely (not from 127.0.0.1), you need to support HTTPS.

Communication Security Matrix:

Communication Type	Protocol	Encryption	Validation
TMDB API	HTTPS/TLS 1.3	End-to-end encryption	Certificate pinning
Remote Addons	HTTPS required	TLS encryption	If an add-on is served via HTTP, CORS headers must be present
IPC Messages	Tauri secure channel	AES-GCM encryption	Message authentication
Local Database	Direct access	SQLCipher encryption	Integrity checks

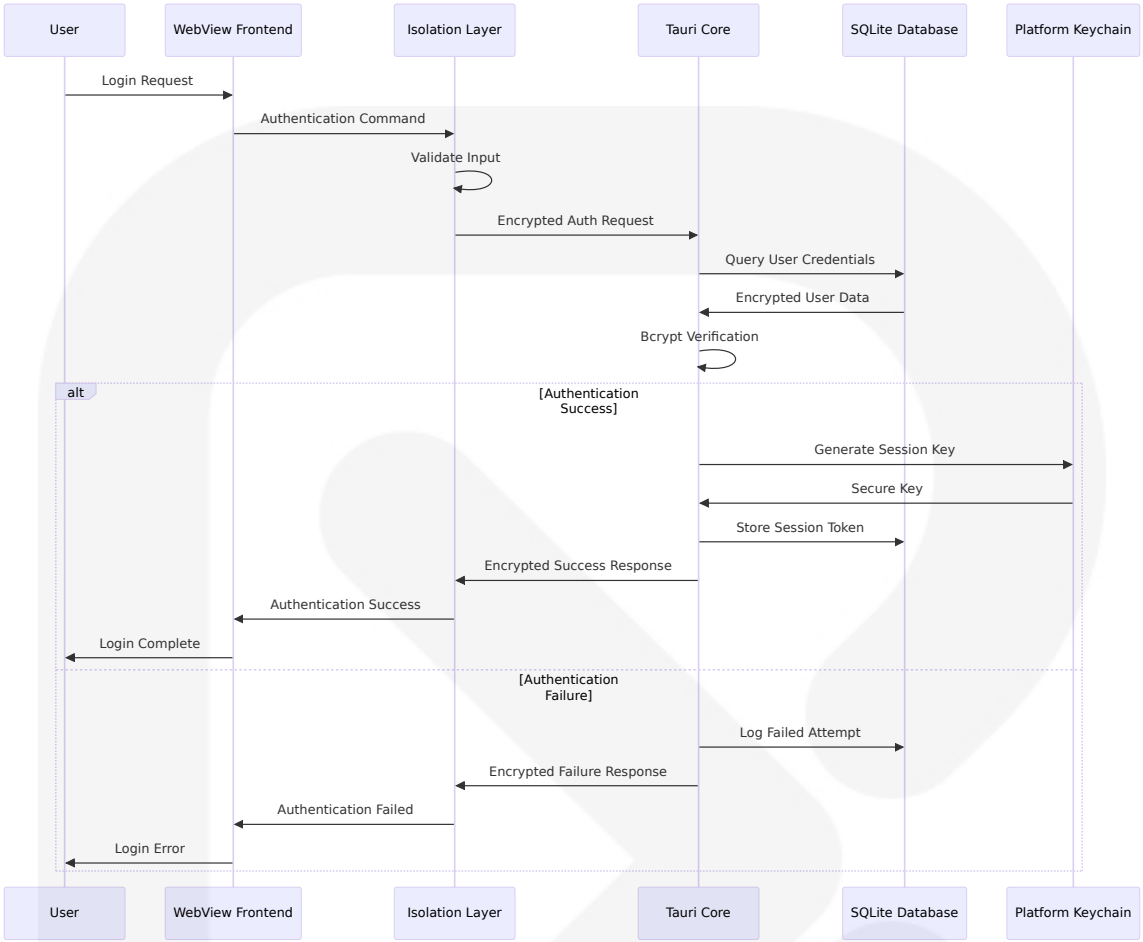
6.4.3.5 Compliance Controls

Data Protection Compliance:

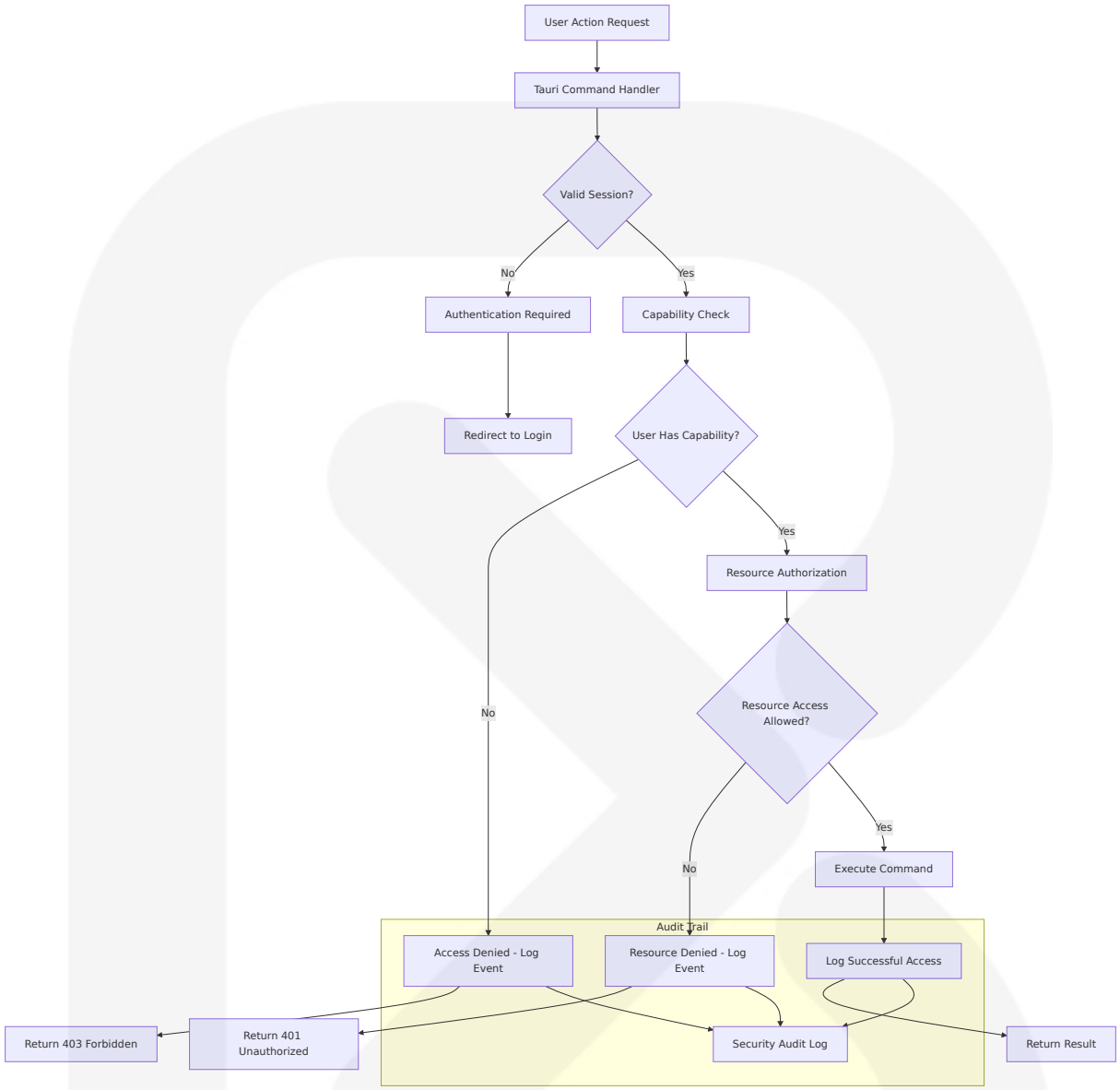
Regulation	Implementation	Controls	Monitoring
GDPR	Data minimization, user consent	Right to deletion, data portability	Audit logging
CCPA	Privacy notices, opt-out mechanisms	Data access requests	Compliance reporting
SOC 2	Security controls documentation	Access controls, encryption	Continuous monitoring

6.4.4 SECURITY ARCHITECTURE DIAGRAMS

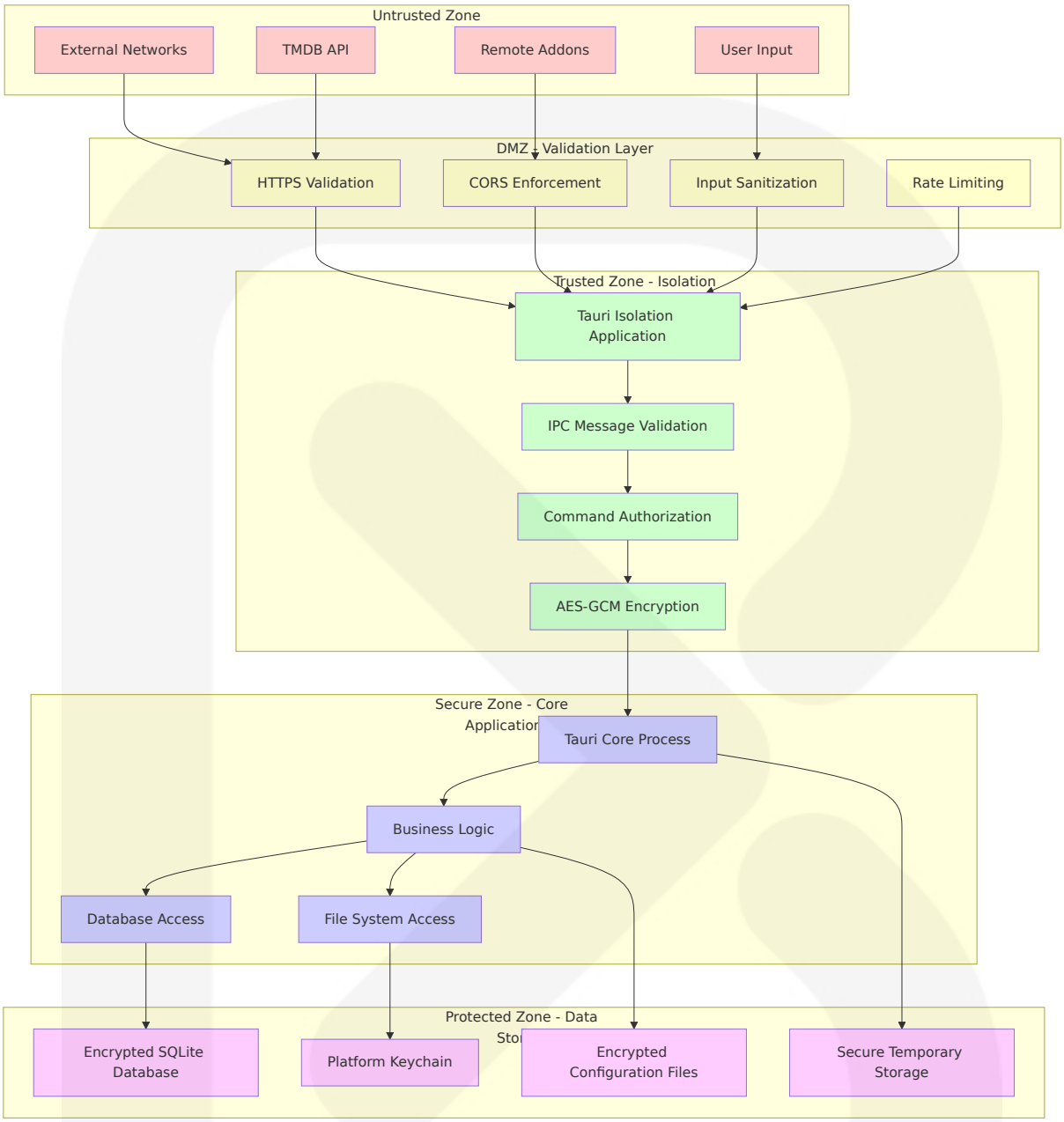
6.4.4.1 Authentication Flow Diagram



6.4.4.2 Authorization Flow Diagram



6.4.4.3 Security Zone Diagram



6.4.5 SECURITY CONTROL MATRICES

6.4.5.1 Access Control Matrix

Resource Type	Guest User	Authenticated User	Admin User	System Process
Content Search	□ Read	□ Read/Write	□ Full Control	□ System Access
User Library	□ None	□ Own Library	□ All Libraries	□ System Access
Addon Management	□ None	□ User Addons	□ System Addons	□ System Access
System Settings	□ None	□ None	□ Full Control	□ System Access

6.4.5.2 Data Classification Matrix

Data Classification	Encryption Required	Access Logging	Retention Period	Backup Required
Public Content Metadata	□ Optional	□ No	30 days	□ No
User Preferences	□ Required	□ Yes	User-controlled	□ Yes
Authentication Data	□ Required	□ Yes	Account lifetime	□ Yes
System Configuration	□ Required	□ Yes	Indefinite	□ Yes

6.4.5.3 Threat Mitigation Matrix

Threat Category	Risk Level	Mitigation Strategy	Implementation	Monitoring
Malicious Addons	High	An add-on in Stremio doesn't generally run on a client's computer. Instead, it is hosted on the Internet just like any we	Remote execution model	Addon health monitoring

Threat Category	Risk Level	Mitigation Strategy	Implementation	Monitoring
		bsite. This brings ease of use and security benefits to the end user		
IPC Attacks	Medium	The Isolation pattern is all about injecting a secure application in between your frontend and Tauri Core to intercept and modify incoming IPC messages	Isolation layer validation	Command audit logging
Data Breaches	High	Multi-layer encryption	AES-256 encryption at rest and in transit	Access pattern analysis
Privilege Escalation	Medium	Capability-based access control	Tauri capabilities system	Permission audit trail

6.4.6 COMPLIANCE REQUIREMENTS

6.4.6.1 Privacy Compliance

GDPR Compliance Implementation:

GDPR Requirement	Implementation	Technical Control	User Interface
Right to Access	Data export functionality	JSON export API	Settings > Privacy > Export Data
Right to Deletion	Account deletion with data purging	Cascading delete triggers	Settings > Account > Delete Account

GDPR Requirement	Implementation	Technical Control	User Interface
Data Portability	Standardized export format	JSON/CSV export	Settings > Privacy > Download Data
Consent Management	Granular consent tracking	Consent database table	Privacy settings dashboard

6.4.6.2 Security Standards Compliance

Security Framework Alignment:

Framework	Compliance Level	Implementation	Validation Method
NIST Cybersecurity Framework	Core Implementation	Risk-based security controls	Annual security assessment
OWASP Top 10	Full Coverage	Secure coding practices	Automated security scanning
ISO 27001	Partial Implementation	Information security management	Internal audit process

6.4.6.3 Platform Security Requirements

Operating System Security Integration:

Platform	Security Feature	Implementation	Benefit
Windows	Windows Defender integration	Automatic malware scanning	Real-time threat protection
macOS	Keychain Services	Secure credential storage	Hardware-backed encryption
Linux	Secret Service API	Encrypted credential storage	Desktop environment integration

This comprehensive security architecture ensures that the Stremio Clone Desktop Application maintains the highest security standards while providing a seamless user experience. The multi-layered approach, combined with Tauri's built-in security features and industry best practices, creates a robust defense against modern cybersecurity threats while maintaining compliance with relevant privacy and security regulations.

6.5 Monitoring and Observability

6.5.1 MONITORING INFRASTRUCTURE

6.5.1.1 Metrics Collection

The Stremio Clone Desktop Application implements a lightweight monitoring infrastructure tailored for desktop applications. Tauri is a framework for building tiny, blazingly fast binaries for all major desktop platforms. The backend of the application is a rust-sourced binary with an API that the front-end can interact with. This architecture enables efficient metrics collection through Rust's robust ecosystem.

Core Metrics Architecture:

Metric Category	Collection Method	Storage	Retention
Application Performance	The Rust tracing crate from the Tokio ecosystem provides a versatile interface for collecting structured telemetry—including metrics, traces, and logs	Local SQLite database	30 days
System Resources	Standard metrics are DB Read/Write speed, CPU, and RAM usage	In-memory aggregation	24 hours
User Interactions	Tauri IPC command tracking	Local storage	7 days

Metric Category	Collection Method	Storage	Retention
External API Performance	HTTP client instrumentation	SQLite with JSONB	14 days

Metrics Collection Implementation:

```
use metrics::{counter, gauge, histogram};
use std::time::Instant;

// Application startup metrics
pub fn track_application_startup() {
    let start_time = Instant::now();

    // Track startup duration
    histogram!("app_startup_duration_seconds",
start_time.elapsed().as_secs_f64());

    // Track successful startup
    counter!("app_startup_total").increment(1);
}

// Content discovery metrics
pub fn track_content_search(query: &str, response_time: f64,
results_count: usize) {
    histogram!("content_search_duration_seconds", response_time);
    gauge!("content_search_results_count", results_count as f64);
    counter!("content_search_total").increment(1);
}

// System resource metrics
pub fn track_system_resources(memory_usage: u64, cpu_usage: f64) {
    gauge!("system_memory_usage_bytes", memory_usage as f64);
    gauge!("system_cpu_usage_percent", cpu_usage);
}
```

6.5.1.2 Log Aggregation

Structured Logging Strategy:

In Rust, the log crate is the de-facto standard for logging. It provides a simple API for logging messages at different levels, including error, warn, info, and debug.

Log Categories and Levels:

Log Category	Level	Destination	Format
Application Events	INFO, DEBUG	Local file	Structured JSON
Error Tracking	ERROR, WARN	Local file + UI notifications	Detailed stack traces
Security Events	WARN, ERROR	Encrypted local storage	Audit trail format
Performance Events	DEBUG	Memory buffer	Metrics format

Logging Configuration:

```
use tracing::{info, warn, error, debug};
use tracing_subscriber::{layer::SubscriberExt,
util::SubscriberInitExt};

pub fn initialize_logging() {
    tracing_subscriber::registry()
        .with(
            tracing_subscriber::EnvFilter::try_from_default_env()
                .unwrap_or_else(|_|
"stremio_clone=debug,tauri=info".into()),
        )
        .with(tracing_subscriber::fmt::layer())
        .init();

    info!("Logging system initialized");
}

// Usage examples
pub fn log_content_discovery(query: &str, results: usize) {
    info!(
        query = %query,
```

```
        results_count = results,
        "Content discovery completed"
    );
}

pub fn log_addon_error(addon_url: &str, error: &str) {
    error!(
        addon_url = %addon_url,
        error = %error,
        "Addon communication failed"
    );
}
```

6.5.1.3 Alert Management

Desktop Application Alert Strategy:

Unlike server applications, desktop applications require user-friendly alerting mechanisms that don't overwhelm the user experience.

Alert Categories:

Alert Type	Trigger Condition	User Notification	Technical Action
Critical Errors	Application crash, data corruption	Modal dialog with recovery options	Automatic crash report generation
Performance Degradation	Response time > 5 seconds	Status bar indicator	Background optimization
Connectivity Issues	API failures > 3 consecutive	Toast notification	Automatic retry with backoff
Storage Issues	Disk space < 100MB	Settings notification	Cache cleanup suggestion

Alert Implementation:

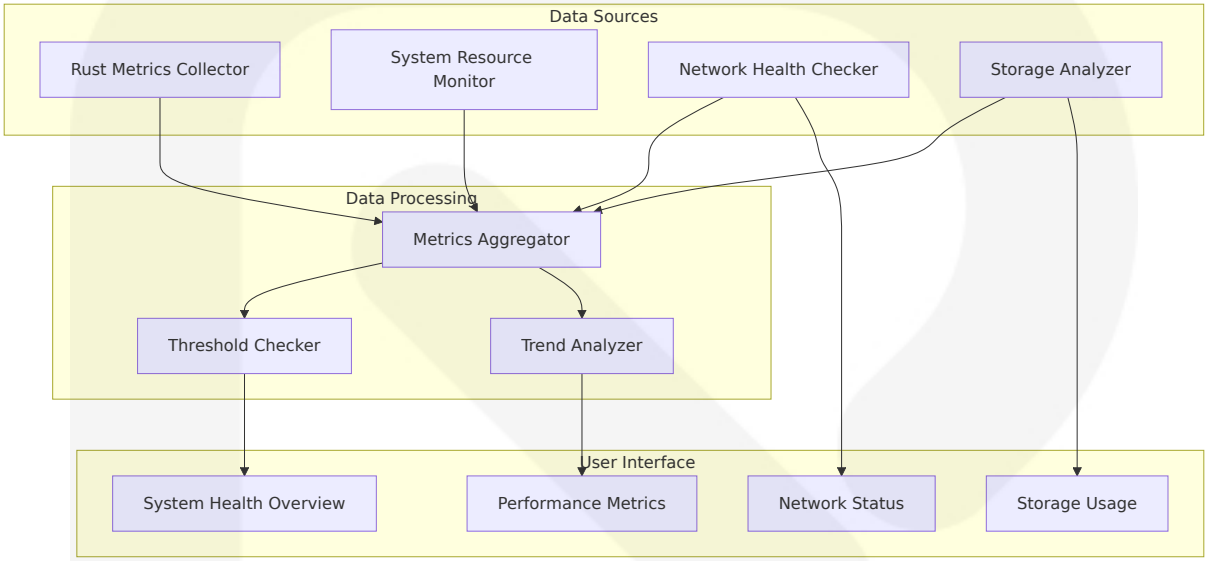
```
use tauri::{Manager, Window};
use serde::Serialize;
```

```rust
#[derive(Serialize)]
struct AlertPayload {
    level: String,
    title: String,
    message: String,
    actions: Vec<String>,
}

pub async fn send_user_alert(
    window: &Window,
    level: &str,
    title: &str,
    message: &str,
) -> Result<(), tauri::Error> {
    let payload = AlertPayload {
        level: level.to_string(),
        title: title.to_string(),
        message: message.to_string(),
        actions: vec!["OK".to_string(), "Details".to_string()],
    };

    window.emit("system-alert", payload)?;
    Ok(())
}

// Performance alert example
pub async fn check_performance_thresholds(window: &Window) {
    let response_time = measure_api_response_time().await;

    if response_time > 5.0 {
        send_user_alert(
            window,
            "warning",
            "Performance Notice",
            "Content loading is slower than usual. Check your internet
connection.",
        ).await.ok();
    }
}
```

6.5.1.4 Dashboard Design

Integrated Monitoring Dashboard:

The application includes a built-in monitoring dashboard accessible through the settings interface, providing users with transparency about application health.



Dashboard Components:

Component	Purpose	Update Frequency	User Benefit
Health Status	Overall application health indicator	Real-time	Quick health assessment
Performance Graph	Response time trends	Every 5 minutes	Performance awareness
Storage Monitor	Cache and database usage	Every hour	Storage management
Network Quality	API connectivity status	Every 30 seconds	Connectivity awareness

6.5.2 OBSERVABILITY PATTERNS

6.5.2.1 Health Checks

Multi-Layer Health Check System:

Microservices-based applications often use heartbeats or health checks to enable their performance monitors, schedulers, and orchestrators to keep track of the multitude of services. If services cannot send some sort of "I'm alive" signal, either on demand or on a schedule, your application might face risks when you deploy updates

Health Check Matrix:

Component	Check Type	Frequency	Success Criteria
Database Connection	Connection test	Every 30 seconds	Query response < 100ms
TMDB API	HTTP health check	Every 60 seconds	Status 200, response < 3s
Addon Services	Manifest validation	Every 5 minutes	Valid JSON response
Local Storage	Disk space check	Every 10 minutes	Available space > 100MB

Health Check Implementation:

```
use std::time::{Duration, Instant};
use sqlx::SqlitePool;

#[derive(Debug, Clone)]
pub struct HealthStatus {
    pub component: String,
    pub status: ComponentStatus,
    pub last_check: Instant,
    pub response_time: Duration,
    pub error_message: Option<String>,
}

#[derive(Debug, Clone)]
pub enum ComponentStatus {
    Healthy,
```

```
    Degraded,
    Unhealthy,
}

pub struct HealthChecker {
    database_pool: SqlitePool,
    tmdb_client: reqwest::Client,
}

impl HealthChecker {
    pub async fn check_database_health(&self) -> HealthStatus {
        let start = Instant::now();

        match sqlx::query("SELECT
1").fetch_one(&self.database_pool).await {
            Ok(_) => HealthStatus {
                component: "database".to_string(),
                status: ComponentStatus::Healthy,
                last_check: Instant::now(),
                response_time: start.elapsed(),
                error_message: None,
            },
            Err(e) => HealthStatus {
                component: "database".to_string(),
                status: ComponentStatus::Unhealthy,
                last_check: Instant::now(),
                response_time: start.elapsed(),
                error_message: Some(e.to_string()),
            },
        }
    }

    pub async fn check_tmdb_health(&self) -> HealthStatus {
        let start = Instant::now();

        match self.tmdb_client
            .get("https://api.themoviedb.org/3/configuration")
            .timeout(Duration::from_secs(5))
            .send()
            .await
        {
            Ok(response) if response.status().is_success() =>
HealthStatus {
```

```
                component: "tmdb_api".to_string(),
                status: ComponentStatus::Healthy,
                last_check: Instant::now(),
                response_time: start.elapsed(),
                error_message: None,
            },
            Ok(response) => HealthStatus {
                component: "tmdb_api".to_string(),
                status: ComponentStatus::Degraded,
                last_check: Instant::now(),
                response_time: start.elapsed(),
                error_message: Some(format!("HTTP {}",
response.status())),
            },
            Err(e) => HealthStatus {
                component: "tmdb_api".to_string(),
                status: ComponentStatus::Unhealthy,
                last_check: Instant::now(),
                response_time: start.elapsed(),
                error_message: Some(e.to_string()),
            },
        }
    }
}
```

6.5.2.2 Performance Metrics

Key Performance Indicators:

Leverage Rust metrics for observability and optimization with Grafana/Prometheus. Metrics provide insights into the system's general performance and specific functionalities. They will also help monitor performance and health.

Performance Metrics Framework:

Metric Type	Examples	Threshold	Action
Response Time	API calls, database queries	< 1 second	Performance optimization

Metric Type	Examples	Threshold	Action
Throughput	Requests per second	> 10 RPS	Capacity planning
Error Rate	Failed requests percentage	< 1%	Error investigation
Resource Usage	Memory, CPU utilization	< 80%	Resource optimization

Performance Monitoring Implementation:

```
use std::collections::HashMap;
use std::sync::{Arc, Mutex};
use std::time::{Duration, Instant};

pub struct PerformanceMonitor {
    metrics: Arc<Mutex<HashMap<String, PerformanceMetric>>>,
}

#[derive(Debug, Clone)]
pub struct PerformanceMetric {
    pub name: String,
    pub value: f64,
    pub timestamp: Instant,
    pub threshold: Option<f64>,
}

impl PerformanceMonitor {
    pub fn new() -> Self {
        Self {
            metrics: Arc::new(Mutex::new(HashMap::new())),
        }
    }

    pub fn record_response_time(&self, operation: &str, duration:
Duration) {
        let mut metrics = self.metrics.lock().unwrap();
        metrics.insert(
            format!("{}_response_time", operation),
            PerformanceMetric {
```

```
                name: operation.to_string(),
                value: duration.as_secs_f64(),
                timestamp: Instant::now(),
                threshold: Some(1.0), // 1 second threshold
            },
        );
    }

    pub fn record_memory_usage(&self, bytes: u64) {
        let mut metrics = self.metrics.lock().unwrap();
        metrics.insert(
            "memory_usage".to_string(),
            PerformanceMetric {
                name: "memory_usage".to_string(),
                value: bytes as f64,
                timestamp: Instant::now(),
                threshold: Some(500_000_000.0), // 500MB threshold
            },
        );
    }

    pub fn get_performance_summary(&self) -> Vec<PerformanceMetric> {
        let metrics = self.metrics.lock().unwrap();
        metrics.values().cloned().collect()
    }
}
```

6.5.2.3 Business Metrics

User Experience Metrics:

Business Metric	Measurement	Target	Impact
Content Discovery Success Rate	Successful searches / Total searches	> 95%	User satisfaction
Addon Installation Success	Successful installs / Attempted installs	> 90%	Feature adoption
Video Playback Success	Successful plays / Attempted plays	> 98%	Core functionality

Business Metric	Measurement	Target	Impact
Application Crash Rate	Crashes / Sessions	< 0.1%	Application stability

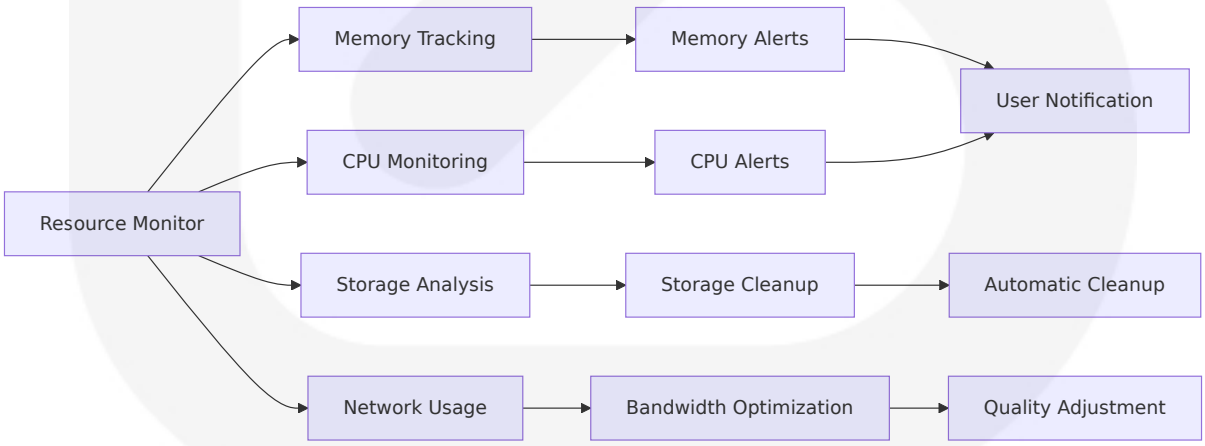
6.5.2.4 SLA Monitoring

Service Level Objectives:

Service Component	Availability Target	Performance Target	Error Rate Target
Application Startup	99.9% successful starts	< 3 seconds	< 0.1% failures
Content Search	99.5% availability	< 1 second response	< 2% failures
Video Playback	99.8% availability	< 5 seconds to start	< 0.5% failures
Addon Management	99.0% availability	< 10 seconds install	< 5% failures

6.5.2.5 Capacity Tracking

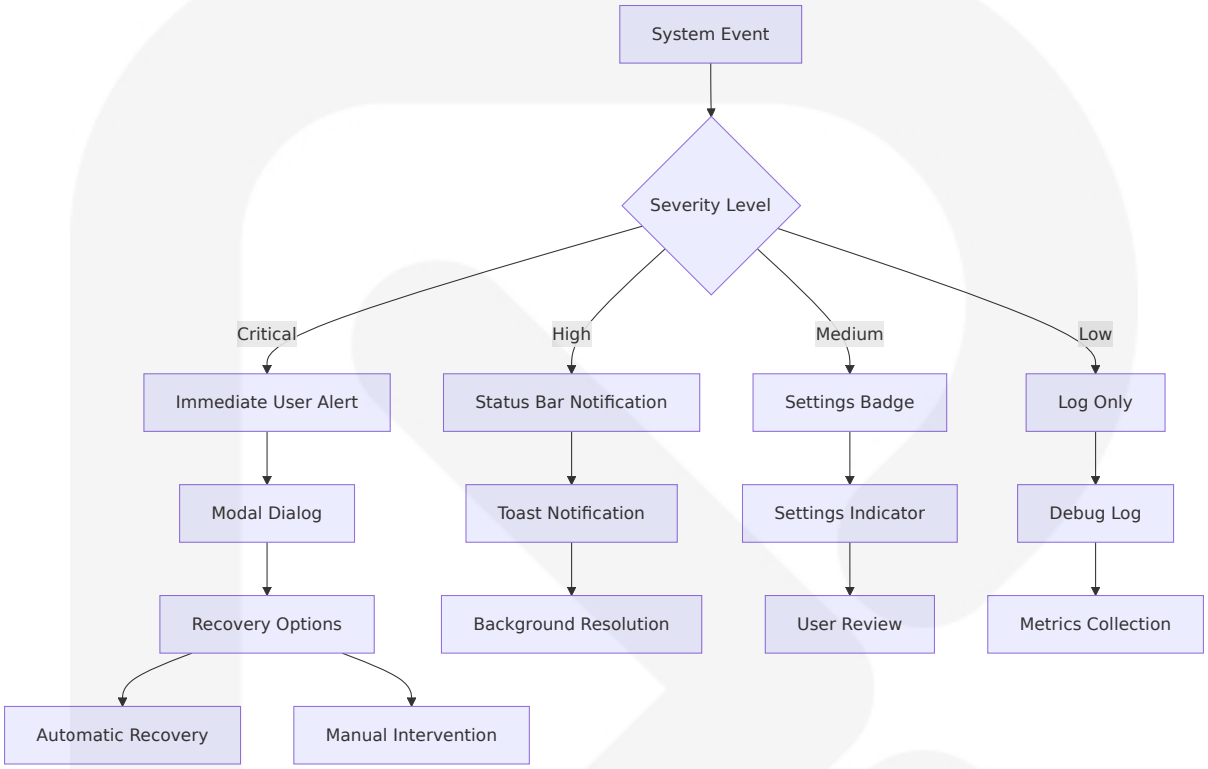
Resource Utilization Monitoring:



6.5.3 INCIDENT RESPONSE

6.5.3.1 Alert Routing

Desktop Application Alert Flow:



Alert Routing Configuration:

Alert Type	Routing Destination	Response Time	Escalation
Application Crash	Immediate modal dialog	Instant	Crash report generation
Network Failure	Toast notification	30 seconds	Retry mechanism
Performance Degradation	Status indicator	5 minutes	Background optimization
Storage Warning	Settings notification	1 hour	Cleanup suggestions

6.5.3.2 Escalation Procedures

Automated Escalation Matrix:

Issue Severity	Initial Response	Escalation Trigger	Escalation Action
Critical	Immediate user notification	No user response in 30s	Automatic recovery attempt
High	Background notification	Issue persists > 5 minutes	User intervention prompt
Medium	Passive notification	Issue persists > 1 hour	Settings recommendation
Low	Log entry only	Pattern detected	Proactive optimization

6.5.3.3 Runbooks

Common Issue Resolution Procedures:

Issue Type	Detection Method	Resolution Steps	Prevention
High Memory Usage	Memory threshold exceeded	1. Clear cache 2. Restart components 3. Notify user	Implement memory limits
API Timeout	Response time > 10s	1. Retry request 2. Switch to cache 3. Show offline mode	Connection pooling
Database Lock	Query timeout	1. Cancel operation 2. Restart transaction 3. Rebuild if needed	WAL mode optimization

Issue Type	Detection Method	Resolution Steps	Prevention
Addon Failure	Health check failure	1. Disable addon 2. Notify user 3. Suggest alternatives	Addon validation

6.5.3.4 Post-Mortem Processes

Incident Analysis Framework:

```
#[derive(Debug, Serialize)]
pub struct IncidentReport {
    pub incident_id: String,
    pub timestamp: DateTime<Utc>,
    pub severity: IncidentSeverity,
    pub component: String,
    pub description: String,
    pub root_cause: Option<String>,
    pub resolution: Option<String>,
    pub prevention_measures: Vec<String>,
}

#[derive(Debug, Serialize)]
pub enum IncidentSeverity {
    Critical,
    High,
    Medium,
    Low,
}

impl IncidentReport {
    pub fn new(component: &str, description: &str) -> Self {
        Self {
            incident_id: uuid::Uuid::new_v4().to_string(),
            timestamp: Utc::now(),
            severity: IncidentSeverity::Medium,
            component: component.to_string(),
            description: description.to_string(),
            root_cause: None,
```

```
            resolution: None,
            prevention_measures: Vec::new(),
        }
    }

    pub async fn save_to_database(&self, pool: &SqlitePool) ->
Result<(), sqlx::Error> {
        sqlx::query!(
            r#"
            INSERT INTO incident_reports
            (incident_id, timestamp, severity, component, description,
root_cause, resolution)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            "#,
            self.incident_id,
            self.timestamp,
            format!("{:?}", self.severity),
            self.component,
            self.description,
            self.root_cause,
            self.resolution
        )
        .execute(pool)
        .await?;

        Ok(())
    }
}
```

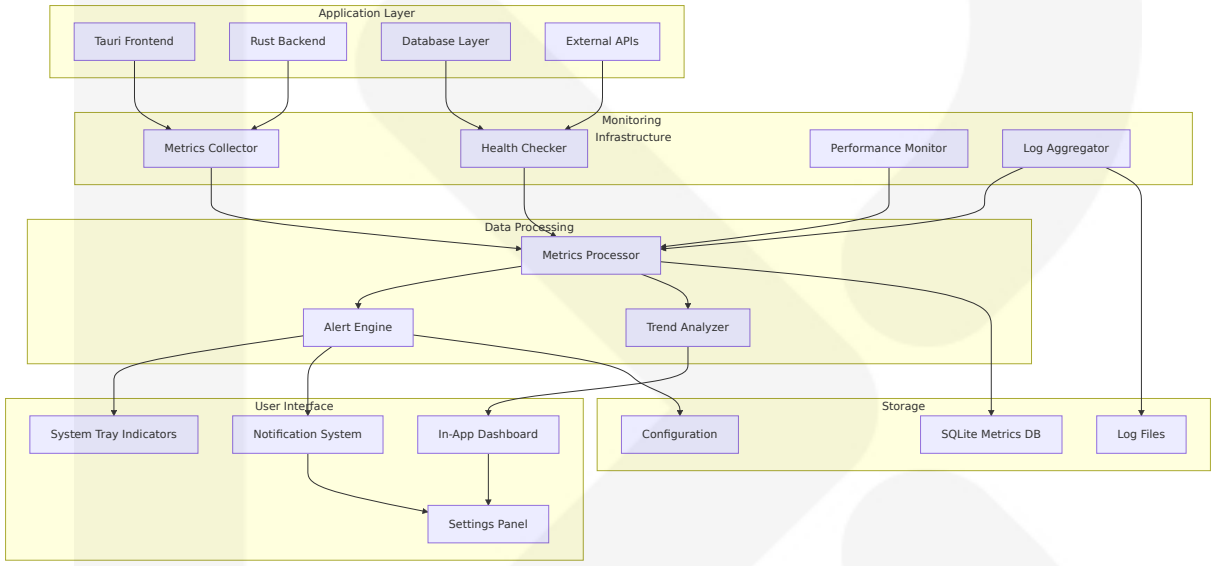
6.5.3.5 Improvement Tracking

Continuous Improvement Metrics:

Improvement Area	Metric	Target	Tracking Method
Mean Time to Detection	Time from issue to alert	< 30 seconds	Automated monitoring
Mean Time to Resolution	Time from alert to fix	< 5 minutes	Incident tracking

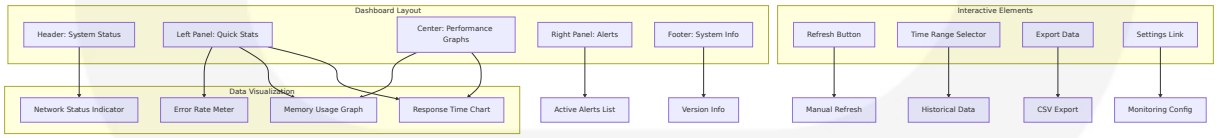
Improvement Area	Metric	Target	Tracking Method
Recurrence Rate	Same issue repeat frequency	< 5%	Pattern analysis
User Impact	Users affected per incident	< 1%	Usage analytics

6.5.4 MONITORING ARCHITECTURE DIAGRAM



6.5.5 PERFORMANCE MONITORING DASHBOARD

Real-Time Monitoring Interface:



6.5.6 MONITORING BEST PRACTICES

Desktop Application Monitoring Guidelines:

Automated checks give you the pulse of your system. A good application health check can run every few minutes, pinging your app's endpoints and confirming everything is responding as expected. These checks often serve as the first line of defense, catching issues before users ever notice them.

Implementation Principles:

Principle	Implementation	Benefit
User-Centric Monitoring	Focus on user experience metrics	Improved user satisfaction
Lightweight Collection	Minimal performance impact	Maintains application speed
Proactive Alerting	Predict issues before they occur	Reduced user disruption
Self-Healing Systems	Automatic recovery mechanisms	Improved reliability

Resource Optimization:

Effective system monitoring and optimization require detailed metrics. This article will teach you how to use metrics in your Rust application to enhance observability, identify and address performance bottlenecks and security issues, and optimize overall efficiency.

The monitoring system is designed to be lightweight and efficient, ensuring that the monitoring overhead doesn't impact the application's performance. All metrics collection and processing are performed asynchronously, and the monitoring data is stored locally to maintain user privacy and reduce external dependencies.

This comprehensive monitoring and observability framework ensures that the Stremio Clone Desktop Application maintains high performance, reliability, and user satisfaction while providing developers with the insights needed for continuous improvement and optimization.

6.6 Testing Strategy

6.6.1 TESTING APPROACH

6.6.1.1 Unit Testing

Testing Framework and Tools:

The Stremio Clone Desktop Application leverages Rust's built-in testing framework combined with Tokio, which is a runtime for writing reliable asynchronous applications with Rust. It provides async I/O, networking, scheduling, timers, and more. For asynchronous testing, Tauri provides support for unit testing and integration testing: Rust unit testing: For the Rust backend, you can write standard Rust unit tests. Create a tests subdirectory under the src-tauri directory and write test files there. Integration testing: Tauri provides a library called tauri-testing for writing integration tests. These tests can be run directly in the simulated Tauri environment without actually building and running the entire application.

Unit Testing Framework Configuration:

Component	Testing Framework	Key Features	Usage Pattern
Rust Backend	Built-in `#[test]` + `#[tokio::test]`	Async testing, time manipulation	Individual function testing
Tauri Commands	`tauri-testing` library	Simulated Tauri environment	IPC command validation
Database Layer	SQLite in-memory testing	Isolated test databases	Data layer verification
External API Clients	Mock HTTP clients with `reqwest-mock`	Request/response simulation	API integration testing

Test Organization Structure:

```rust
// Example unit test structure for Rust backend
#[cfg(test)]
mod tests {
    use super::*;
    use tokio_test;

    #[tokio::test]
    async fn test_content_search() {
        let search_service = ContentSearchService::new_mock();
        let results = search_service.search("test
query").await.unwrap();
        assert!(!results.is_empty());
    }

    #[test]
    fn test_addon_manifest_validation() {
        let manifest = AddonManifest {
            id: "test.addon".to_string(),
            name: "Test Addon".to_string(),
            // ... other fields
        };
        assert!(validate_addon_manifest(&manifest).is_ok());
    }
}
```

Mocking Strategy:

Mock Type	Implementation	Purpose	Test Coverage
TMDB API	`mockito` or `wiremock`	External API simulation	Content discovery logic
Addon Services	HTTP mock servers	Remote addon testing	Addon communication
Database	In-memory SQLite	Data persistence testing	Database operations
File System	`tempfile` crate	File operations testing	Configuration management

Code Coverage Requirements:

Cargo subcommand to easily use LLVM source-based code coverage. This is a wrapper around rustc -C instrument-coverage and provides: Generate very precise coverage data. (line, region, and branch coverage. branch coverage is currently optional and requires nightly, see #8 for more) Support cargo test, cargo run, and cargo nextest with command-line interface compatible with cargo.

Coverage Type	Target Threshold	Measurement Tool	Reporting Format
Line Coverage	85% minimum	`cargo-llvm-cov`	HTML + LCOV
Branch Coverage	80% minimum	`cargo-llvm-cov` (nightly)	Console + CI reports
Function Coverage	90% minimum	Built-in coverage tools	Summary reports

Test Naming Conventions:

```
// Test naming pattern: test_[component]_[scenario]_[expected_outcome]
#[tokio::test]
async fn test_content_search_valid_query_returns_results() { }

#[test]
fn test_addon_validation_invalid_manifest_returns_error() { }

#[tokio::test(start_paused = true)]
async fn test_rate_limiter_exceeds_limit_delays_request() { }
```

Test Data Management:

Data Type	Management Strategy	Storage Location	Cleanup Method
Test Fixtures	JSON files in `tests/fixtures/`	Version controlled	Automatic
Mock Responses	Embedded in test code	In-memory	Test completion

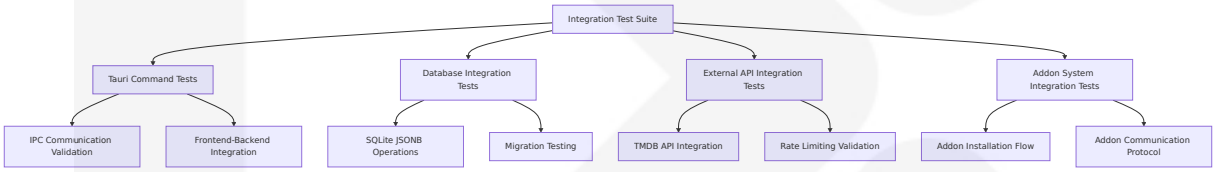
Data Type	Management Strategy	Storage Location	Cleanup Method
Temporary Files	`tempfile` crate	System temp directory	Automatic cleanup
Test Databases	In-memory SQLite	Memory only	Process termination

6.6.1.2 Integration Testing

Service Integration Test Approach:

Integration testing focuses on verifying the interaction between different system components, particularly the Tauri IPC layer, database operations, and external service integrations.

Integration Test Architecture:



API Testing Strategy:

API Category	Testing Approach	Mock Strategy	Validation Points
TMDB API	Live API + Mock fallback	HTTP request/response mocking	Rate limits, data format, error handling
Tauri Commands	Direct command invocation	Simulated frontend calls	Parameter validation, response format
Addon APIs	Mock addon servers	HTTPS mock endpoints	CORS headers, manifest validation

Database Integration Testing:

```rust
// Example database integration test
#[tokio::test]
async fn test_content_metadata_storage_retrieval() {
    let pool = create_test_database().await;

    // Test JSONB storage
    let metadata = ContentMetadata {
        title: "Test Movie".to_string(),
        // ... other fields
    };

    store_content_metadata(&pool, &metadata).await.unwrap();
    let retrieved = get_content_metadata(&pool,
metadata.id).await.unwrap();

    assert_eq!(retrieved.title, metadata.title);
}
```

External Service Mocking:

Service	Mock Implementation	Test Scenarios	Error Simulation
TMDB API	`wiremock` HTTP server	Success, rate limit, timeout	429, 500, network errors
Remote Addons	Mock HTTPS endpoints	Valid/invalid manifests	CORS failures, SSL errors
System Services	Platform-specific mocks	File system, network	Permission denied, disk full

Test Environment Management:

```rust
// Test environment setup
pub struct TestEnvironment {
    pub database: SqlitePool,
    pub mock_server: MockServer,
    pub temp_dir: TempDir,
}

impl TestEnvironment {
```

```
    pub async fn new() -> Self {
        let database = create_in_memory_database().await;
        let mock_server = MockServer::start().await;
        let temp_dir = TempDir::new().unwrap();

        Self { database, mock_server, temp_dir }
    }
}
```

6.6.1.3 End-to-End Testing

E2E Test Scenarios:

Tauri also provides support for end-to-end testing support utilizing the WebDriver protocol. Both desktop and mobile work with it, except for macOS which does not provide a desktop WebDriver client. See more about WebDriver support here.

WebDriver Testing Architecture:

WebDriver is a standardized interface to interact with web documents primarily intended for automated testing. Tauri supports the WebDriver interface by leveraging the native platform's WebDriver server underneath a cross-platform wrapper tauri-driver. On desktop, only Windows and Linux are supported due to macOS not having a WKWebView driver tool available.

Platform	WebDriver Implementation	Requirements	Limitations
Windows	Microsoft Edge WebDriver	`msedgedriver.exe` in PATH	Version matching required
Linux	WebKitWebDriver	`webkit2gtk-driver` package	Distribution-specific packages
macOS	Not supported	N/A	No WKWebView driver available

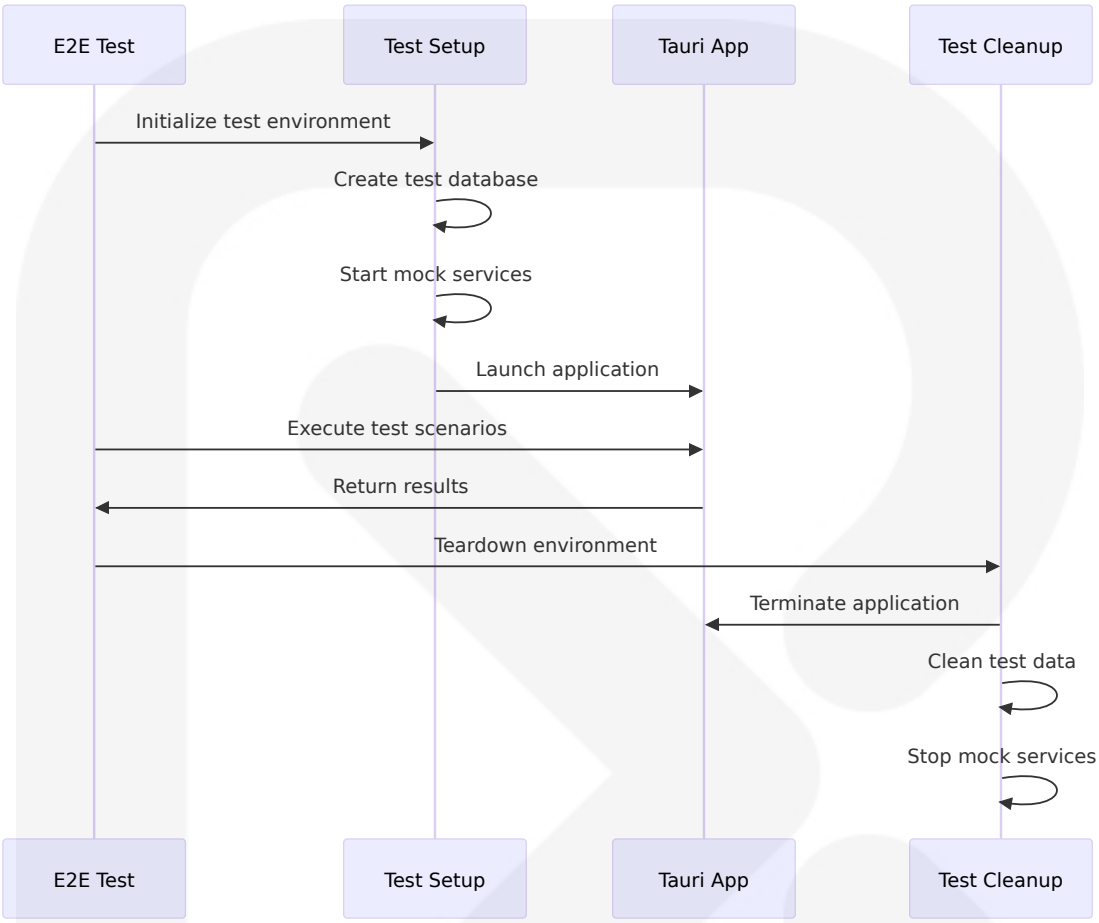
E2E Test Implementation:

```rust
// Example E2E test using WebDriver
use thirtyfour::prelude::*;

#[tokio::test]
async fn test_complete_user_workflow() {
    let caps = DesiredCapabilities::new();
    caps.set_browser_name("wry")?;
    caps.set_capability("tauri:options", json!({
        "application": "/path/to/stremio-clone-binary"
    }))?;

    let driver = WebDriver::new("http://localhost:4444", caps).await?;

    // Test user authentication
    let login_button = driver.find(By::Id("login-button")).await?;
    login_button.click().await?;

    // Test content search
    let search_input = driver.find(By::Id("search-input")).await?;
    search_input.send_keys("test movie").await?;

    // Verify search results
    let results = driver.find_all(By::ClassName("search-
result")).await?;
    assert!(!results.is_empty());

    driver.quit().await?;
}
```

UI Automation Approach:

Test Category	Automation Strategy	Tools Used	Coverage Areas
User Workflows	WebDriver automation	`thirtyfour` crate	Complete user journeys
UI Components	Element interaction testing	Selenium-compatible tools	Form validation, navigation
Cross-Platform	Platform-specific drivers	`tauri-driver` wrapper	Windows/Linux compatibility

Test Data Setup/Teardown:



Performance Testing Requirements:

Performance Metric	Target Value	Test Method	Failure Threshold
Application Startup	< 3 seconds	Automated timing	> 5 seconds
Search Response	< 1 second	WebDriver timing	> 3 seconds
Video Stream Start	< 5 seconds	Media player integration	> 10 seconds
Memory Usage	< 200MB baseline	System monitoring	> 500MB

Cross-Browser Testing Strategy:

Since Tauri applications use system WebViews, cross-browser testing focuses on different system WebView implementations rather than traditional browsers.

Platform	WebView Engine	Testing Approach	Compatibility Notes
Windows	WebView2 (Chromium)	Edge WebDriver testing	Modern Chromium features
Linux	WebKitGTK	WebKit driver testing	WebKit-specific behaviors
macOS	WKWebView	Manual testing only	No automated driver available

6.6.2 TEST AUTOMATION

CI/CD Integration:

The testing strategy integrates with continuous integration pipelines to ensure comprehensive automated testing across all supported platforms.

```
# Example GitHub Actions workflow
name: Test Suite
on: [push, pull_request]

jobs:
  test:
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
    runs-on: ${{ matrix.os }}

    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable

      # Install coverage tools
      - name: Install cargo-llvm-cov
        run: cargo install cargo-llvm-cov
```

```
      # Run unit tests with coverage
      - name: Run unit tests
        run: cargo llvm-cov --workspace --lcov --output-path lcov.info

      # Run integration tests
      - name: Run integration tests
        run: cargo test --test integration_tests

      # E2E tests (Linux and Windows only)
      - name: Run E2E tests
        if: matrix.os != 'macos-latest'
        run: |
          # Install WebDriver dependencies
          cargo install tauri-driver
          # Run E2E test suite
          cargo test --test e2e_tests
```

Automated Test Triggers:

Trigger Event	Test Suite	Execution Time	Failure Action
Pull Request	Unit + Integration	5-10 minutes	Block merge
Main Branch Push	Full test suite + E2E	15-20 minutes	Notify team
Nightly Build	Performance + Security tests	30-45 minutes	Generate report
Release Tag	Complete validation suite	45-60 minutes	Block release

Parallel Test Execution:

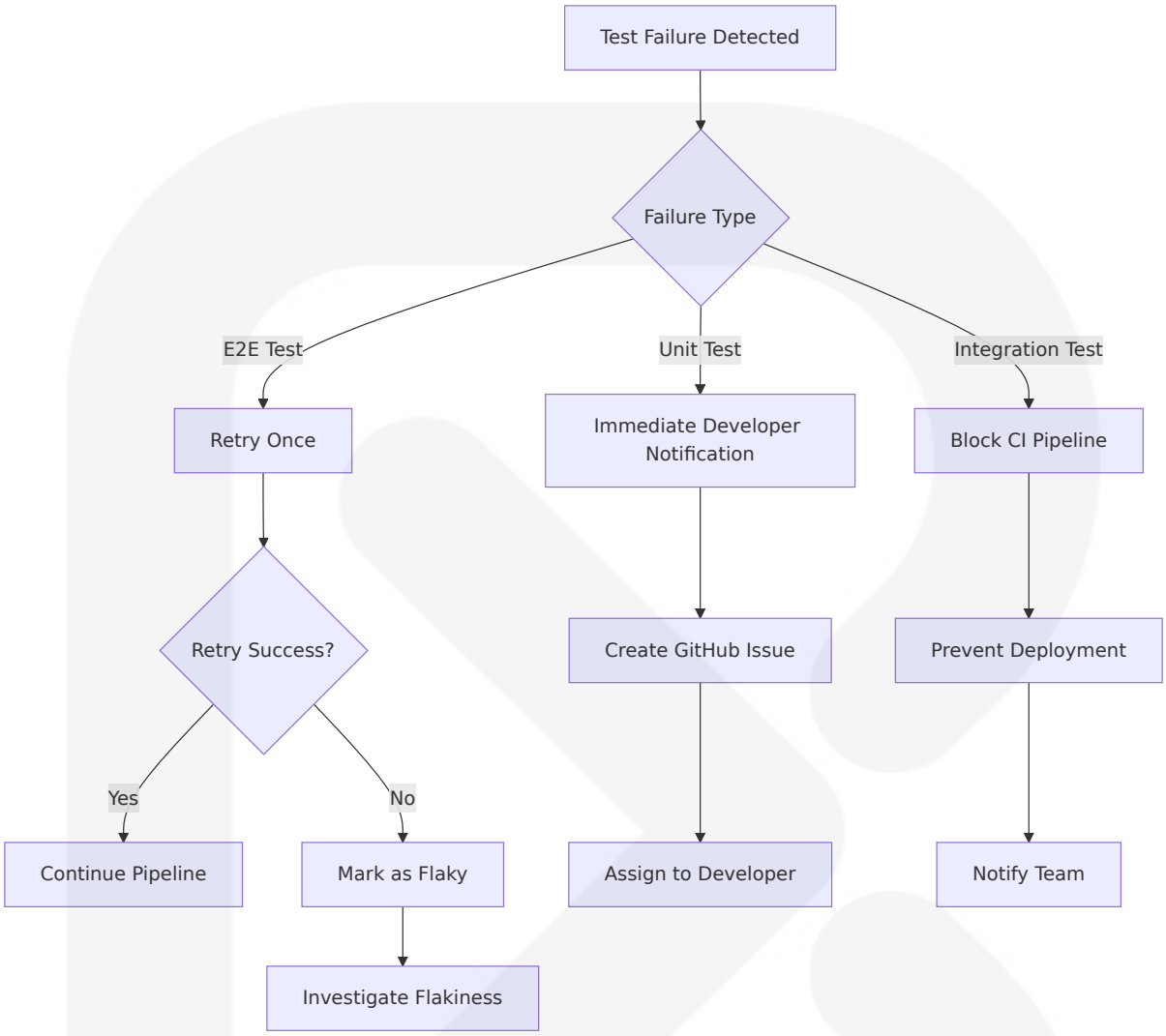
To use the multi-threaded runtime, the macro can be configured using · #[tokio::test(flavor = "multi_thread", worker_threads = 1)] async fn my_test() { assert!(true); } The worker_threads option configures the number of worker threads, and defaults to the number of cpus on the system. Note: The multi-threaded runtime requires the rt-multi-thread feature flag.

```
// Parallel test configuration
[profile.test]
opt-level = 1  # Faster test compilation
debug = true   # Debug info for better error messages

#### Test execution configuration
[workspace.metadata.test]
parallel = true
worker_threads = 4
timeout = "300s"
```

Test Reporting Requirements:

Report Type	Format	Audience	Frequency
Coverage Reports	HTML + LCOV	Developers	Per commit
Performance Reports	JSON + Charts	Team leads	Daily
E2E Test Results	JUnit XML	CI/CD system	Per build
Security Test Results	SARIF	Security team	Weekly

Failed Test Handling:



Flaky Test Management:

Flaky Test Category	Detection Method	Mitigation Strategy	Resolution Timeline
Timing-dependent	Multiple run analysis	For unit tests, it is often useful to run with paused time throughout. This can be achieved simply by setting the macro argument start_paused to true: #[tokio::test(start_paused = true)] async fn paused_time() { let start = std::time::Instant::now(); tokio::time::sleep(Duration::from_millis(500)).await; println!("{:?}ms", start.elapsed	1 week

Flaky Test Category	Detection Method	Mitigation Strategy	Resolution Timeline
		().as_millis()); } Keep in mind that the start_paused attribute requires the tokio feature test-util.	
Resource-dependent	Environment monitoring	Resource isolation, cleanup	3 days
Network-dependent	Mock service failures	Better mocking, retry logic	1 week
Platform-specific	Cross-platform analysis	Platform-specific test variants	2 weeks

6.6.3 QUALITY METRICS

Code Coverage Targets:

Cargo subcommand to easily use LLVM source-based code coverage. This is a wrapper around rustc -C instrument-coverage and provides: Generate very precise coverage data. (line, region, and branch coverage. branch coverage is currently optional and requires nightly, see #8 for more) Support cargo test, cargo run, and cargo nextest with command-line interface compatible with cargo.

Coverage Type	Minimum Target	Ideal Target	Measurement Tool	Reporting
Line Coverage	85%	90%	`cargo-llvm-cov`	HTML dashboard
Branch Coverage	80%	85%	`cargo-llvm-cov` (nightly)	CI reports
Function Coverage	90%	95%	Built-in tools	Summary metrics

Coverage Type	Minimum Target	Ideal Target	Measurement Tool	Reporting
Integration Coverage	75%	80%	Combined test runs	Weekly reports

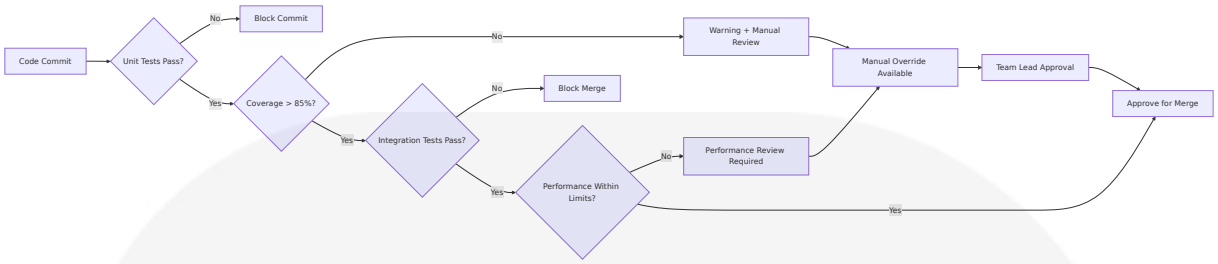
Test Success Rate Requirements:

Test Category	Success Rate Target	Measurement Period	Action Threshold
Unit Tests	99.5%	Per commit	< 98% triggers investigation
Integration Tests	98%	Daily	< 95% blocks deployment
E2E Tests	95%	Weekly	< 90% requires immediate action
Performance Tests	90%	Monthly	< 85% triggers optimization

Performance Test Thresholds:

Performance Metric	Baseline	Warning Threshold	Critical Threshold	Test Frequency
Application Startup	2 seconds	3 seconds	5 seconds	Every build
Memory Usage	150MB	200MB	300MB	Continuous
API Response Time	500ms	1 second	2 seconds	Every test run
Database Query Time	10ms	50ms	100ms	Per query test

Quality Gates:

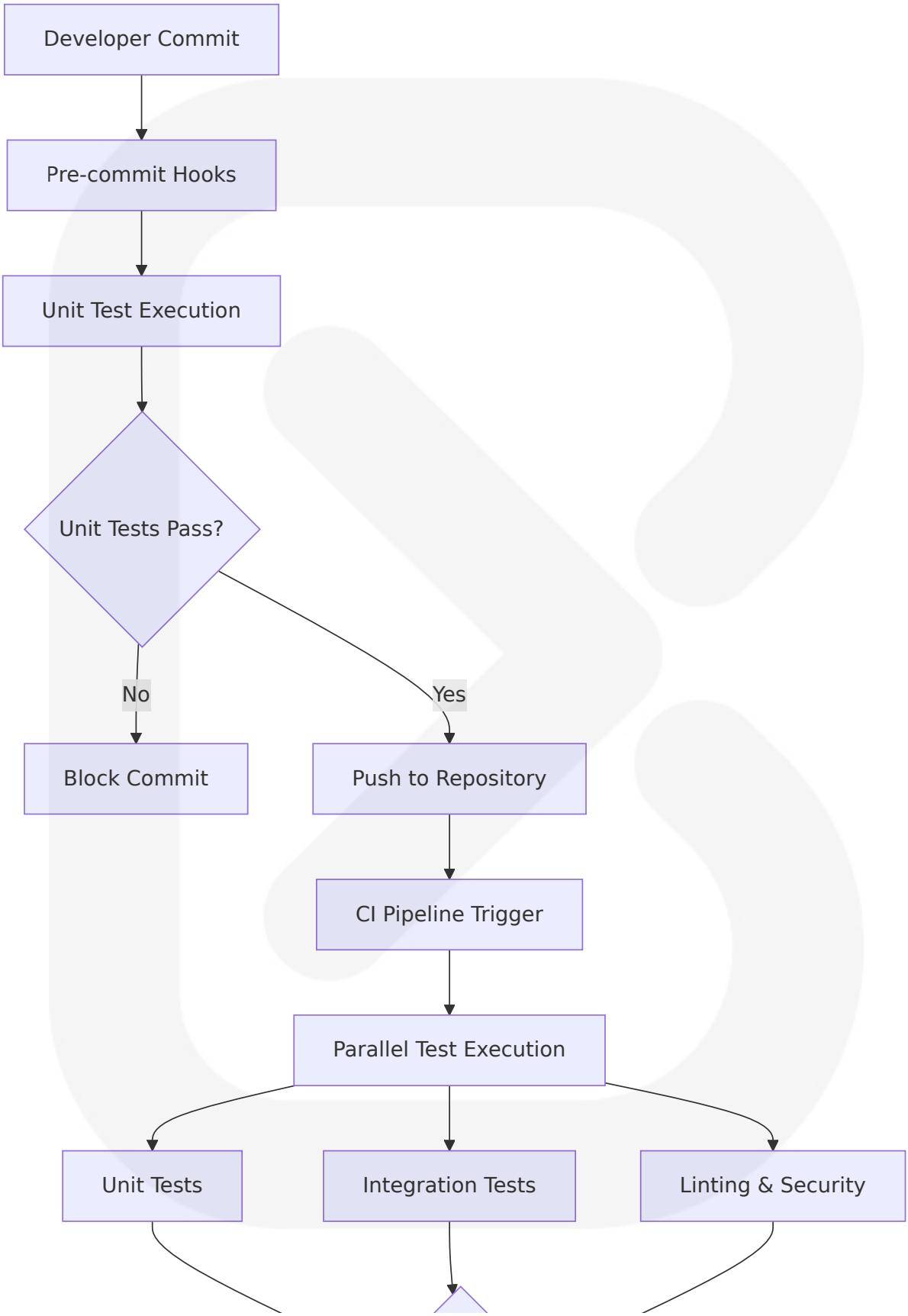


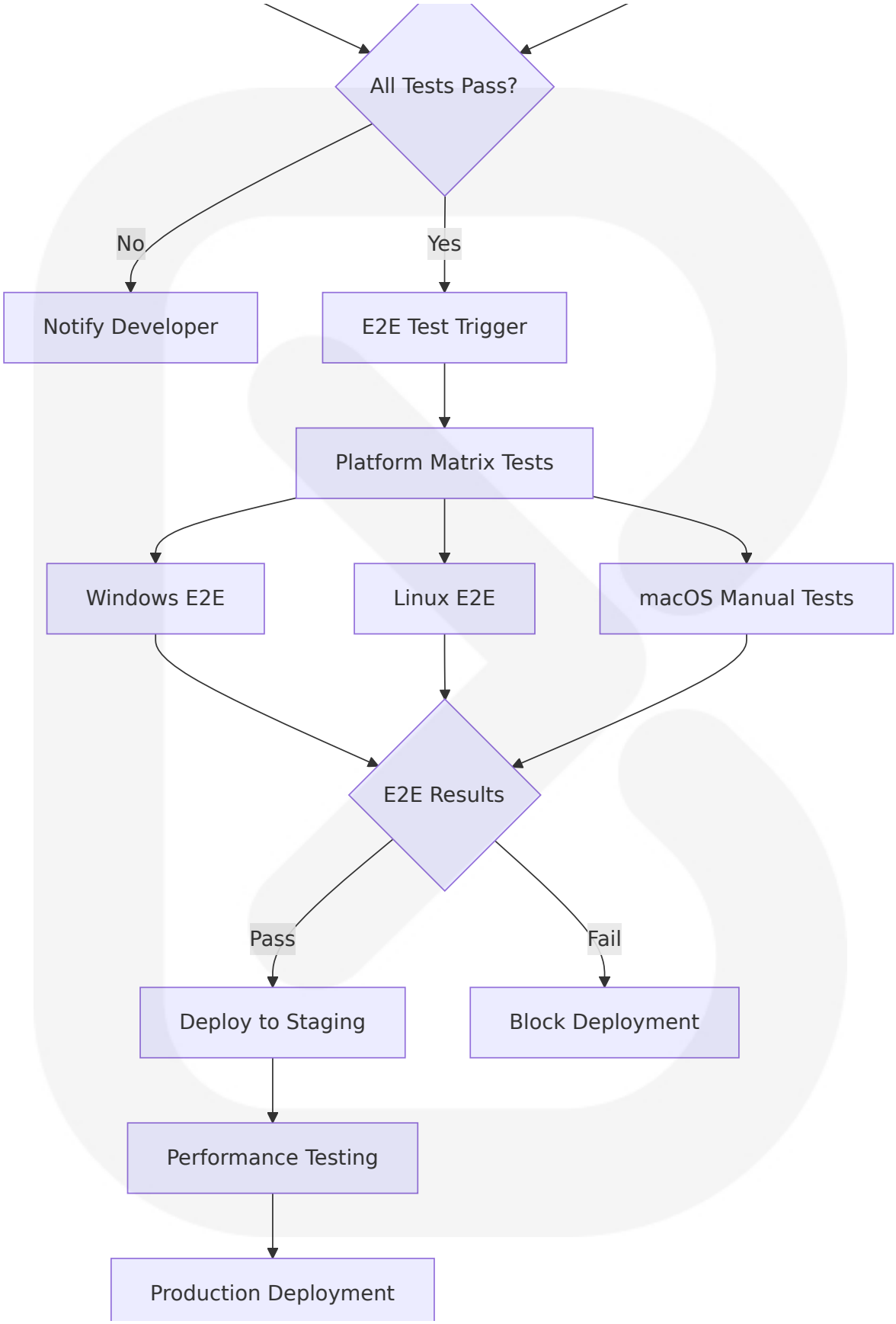
Documentation Requirements:

Documentation Type	Coverage Requirement	Update Frequency	Quality Check
API Documentation	100% of public APIs	Per API change	Automated doc tests
Test Documentation	All complex test scenarios	Per test addition	Manual review
Setup Instructions	Complete environment setup	Per dependency change	CI validation
Troubleshooting Guides	Common test failures	Monthly	Team validation

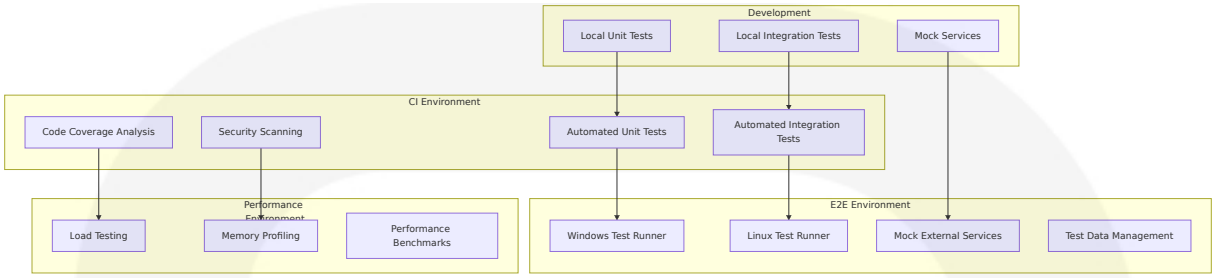
6.6.4 TEST EXECUTION FLOW

Test Execution Architecture:

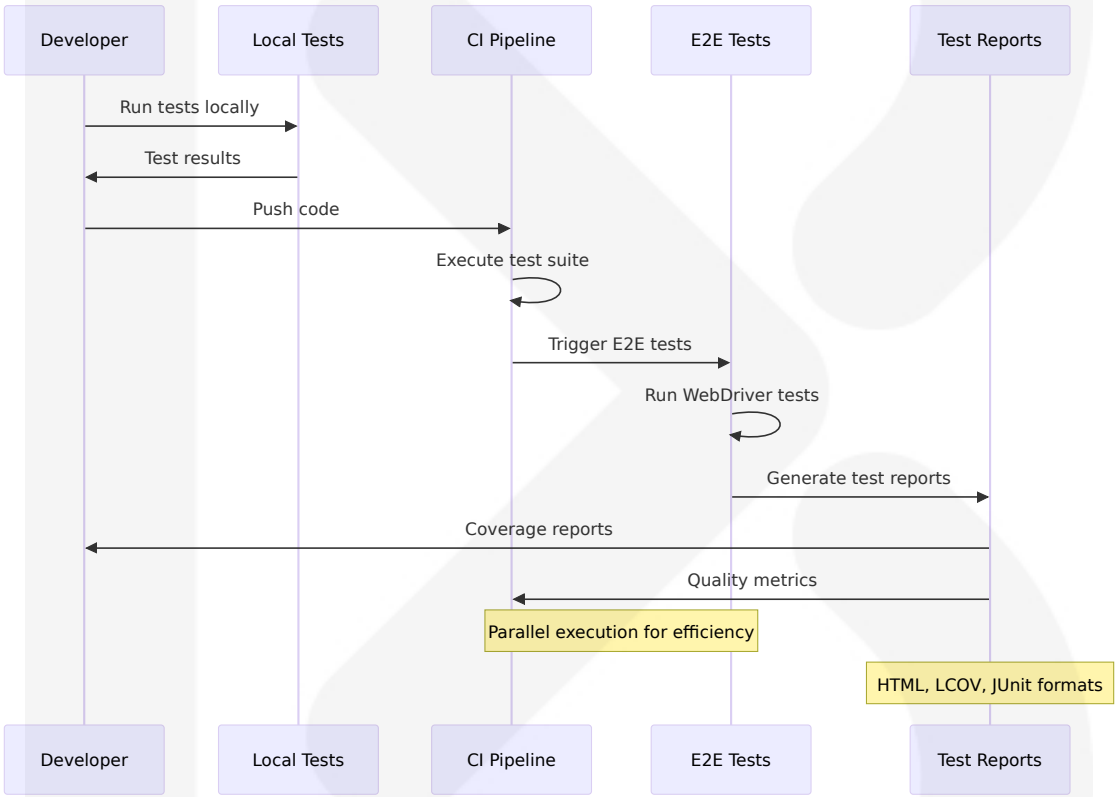

Developer Commit
Pre-commit Hooks
Unit Test Execution
Unit Tests Pass?
No
Yes
Block Commit
Push to Repository
CI Pipeline Trigger
Parallel Test Execution
Unit Tests
Integration Tests
Linting & Security


All Tests Pass?
No
Yes
Notify Developer
E2E Test Trigger
Platform Matrix Tests
Windows E2E
Linux E2E
macOS Manual Tests
E2E Results
Pass
Fail
Deploy to Staging
Block Deployment
Performance Testing
Production Deployment

Test Environment Architecture:



Test Data Flow:



This comprehensive testing strategy ensures the Stremio Clone Desktop Application maintains high quality, reliability, and performance across all supported platforms while leveraging the latest Rust testing tools and Tauri-specific testing capabilities. The strategy balances thorough testing coverage with practical execution time constraints, providing robust quality assurance for the desktop media center application.

7. User Interface Design

7.1 CORE UI TECHNOLOGIES

7.1.1 Frontend Technology Stack

The Stremio Clone Desktop Application leverages Tauri 2.0's ability to integrate any frontend framework that compiles to HTML, JavaScript, and CSS for building their user experience. In a Tauri application the frontend is written in your favorite web frontend stack and renders using WRY, a library which provides a unified interface to the system webview, leveraging WKWebView on macOS & iOS, WebView2 on Windows, WebKitGTK on Linux and Android System WebView on Android.

Primary UI Technologies:

Technology	Version	Purpose	Platform Support
System WebView	Platform Native	UI Rendering Engine	Microsoft Edge WebView 2 on Windows, WKWebView on macOS and webkit gtk on Linux
HTML5	Latest Standard	Semantic markup and structure	All supported platforms
CSS3	Latest Standard	Styling, animations, responsive design	All supported platforms
JavaScript/TypeScript	ES2022+	Dynamic interactions and logic	All supported platforms
Frontend Framework	Framework-agnostic	Component-based UI development	Templates for vanilla, Vue.js, Svelte, React, SolidJ

Technology	Version	Purpose	Platform Support
			S, Angular, Preact, Yew, Leptos, and Sycamore

Recommended Frontend Stack:

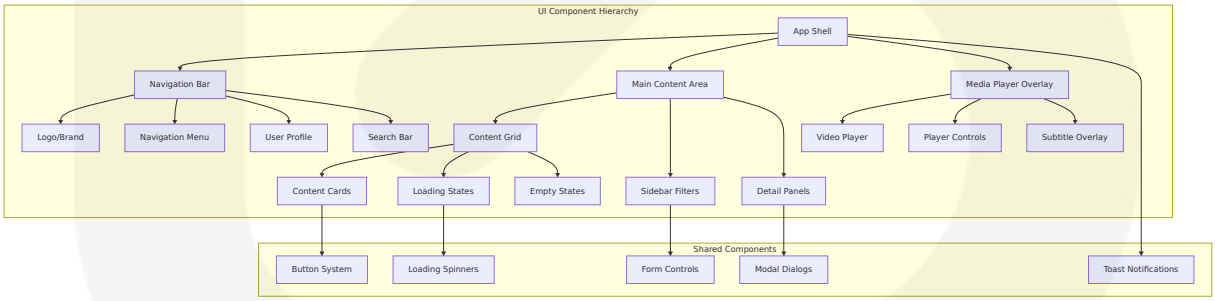
Based on the media center requirements and performance considerations, the recommended stack includes:

- **Svelte/SvelteKit**: Optimal for performance with minimal bundle size
- **Tailwind CSS**: Utility-first CSS framework for rapid UI development
- **Lucide Icons**: Consistent iconography system
- **Video.js**: Web-based video player for media playback

7.1.2 UI Component Architecture

Component-Based Design System:

The application follows a modular component architecture inspired by modern design systems, utilizing components-based UI design with shadcn/ui, Radix UI for UI primitives, native-looking window controls with tauri-controls, and support for dark and light modes.



7.1.3 Styling and Theming System

Design Token Architecture:

Token Category	Implementation	Values	Usage
Colors	CSS Custom Properties	Primary, secondary, accent, semantic colors	Background, text, borders, states
Typography	Font system with scale	Inter/System fonts, 6-level scale	Headings, body text, captions
Spacing	8px grid system	4px, 8px, 16px, 24px, 32px, 48px, 64px	Margins, padding, gaps
Shadows	Layered elevation system	4 elevation levels	Cards, modals, dropdowns
Border Radius	Consistent rounding	4px, 8px, 12px, 16px	Cards, buttons, inputs

Theme Configuration:

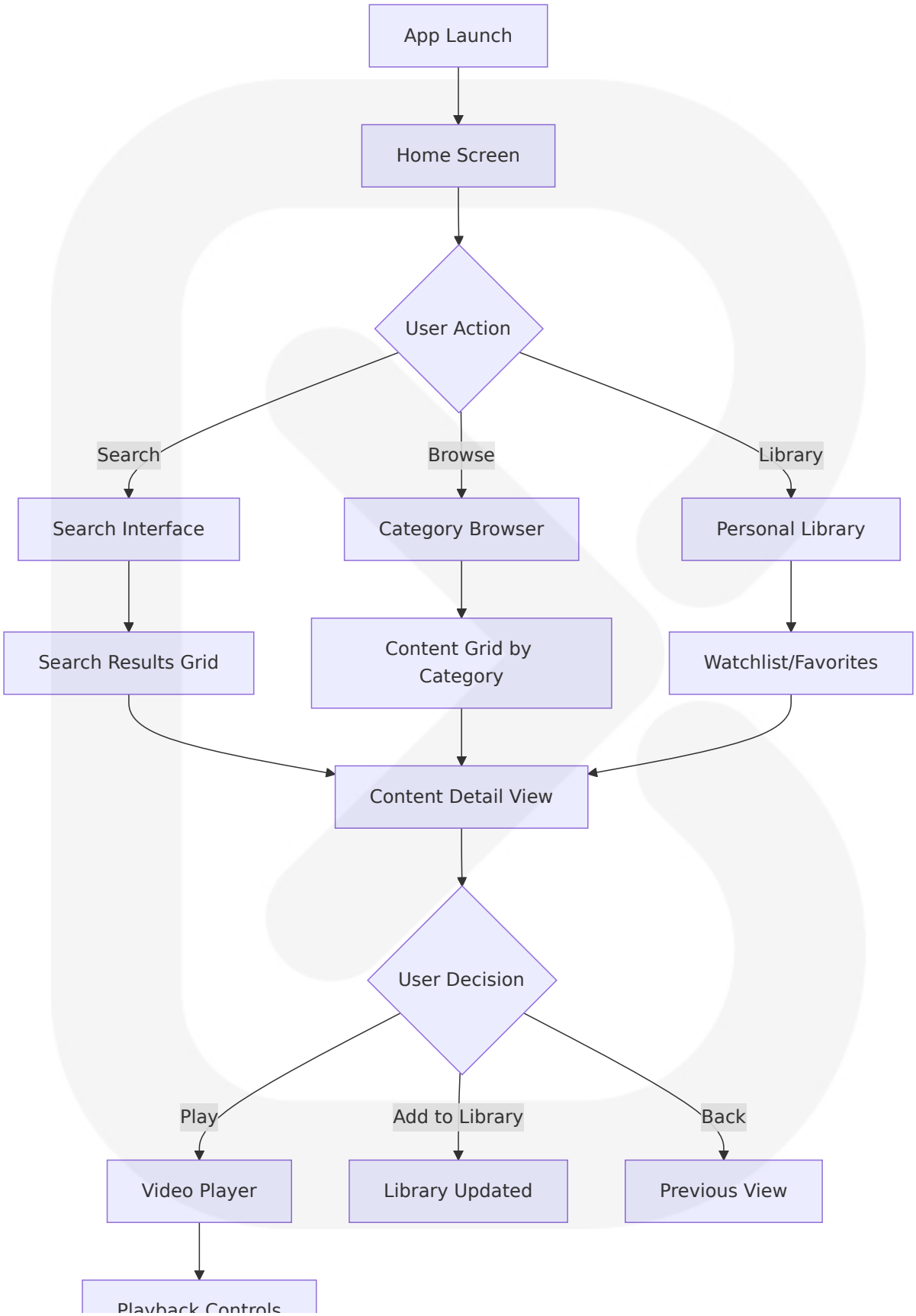
```
:root {
  /* Color Palette - Dark Theme (Primary) */
  --color-background: #0a0a0a;
  --color-surface: #1a1a1a;
  --color-surface-elevated: #2a2a2a;
  --color-primary: #8b5cf6;
  --color-primary-hover: #7c3aed;
  --color-text-primary: #ffffff;
  --color-text-secondary: #a1a1aa;
  --color-text-muted: #71717a;
  --color-border: #27272a;
  --color-accent: #f59e0b;

  /* Typography Scale */
  --font-family-primary: 'Inter', -apple-system, BlinkMacSystemFont,
system-ui, sans-serif;
  --font-size-xs: 0.75rem;
  --font-size-sm: 0.875rem;
  --font-size-base: 1rem;
  --font-size-lg: 1.125rem;
  --font-size-xl: 1.25rem;
  --font-size-2xl: 1.5rem;
  --font-size-3xl: 1.875rem;
```

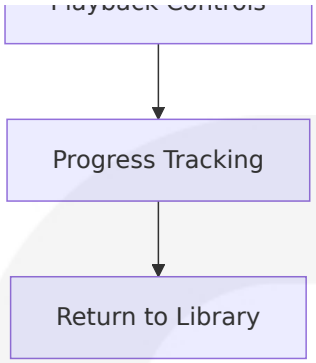
```
  /* Spacing System */
  --spacing-1: 0.25rem;
  --spacing-2: 0.5rem;
  --spacing-3: 0.75rem;
  --spacing-4: 1rem;
  --spacing-6: 1.5rem;
  --spacing-8: 2rem;
  --spacing-12: 3rem;
  --spacing-16: 4rem;
}

[data-theme="light"] {
  --color-background: #ffffff;
  --color-surface: #f8fafc;
  --color-surface-elevated: #ffffff;
  --color-text-primary: #0f172a;
  --color-text-secondary: #475569;
  --color-text-muted: #64748b;
  --color-border: #e2e8f0;
}
```

7.2 UI USE CASES

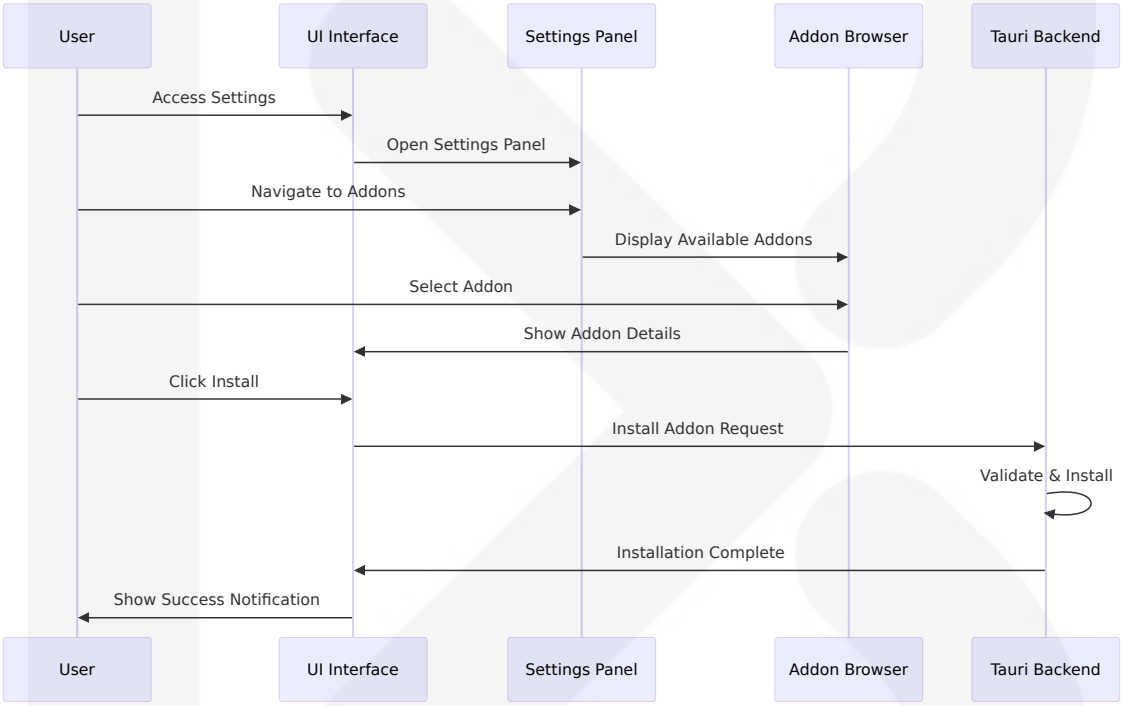
7.2.1 Primary User Workflows

Content Discovery Journey:


App Launch
Home Screen
User Action
Search
Browse
Library
Search Interface
Category Browser
Personal Library
Search Results Grid
Content Grid by Category
Watchlist/Favorites
Content Detail View
User Decision
Play
Add to Library
Back
Video Player
Library Updated
Previous View
Playback Controls



Addon Management Workflow:



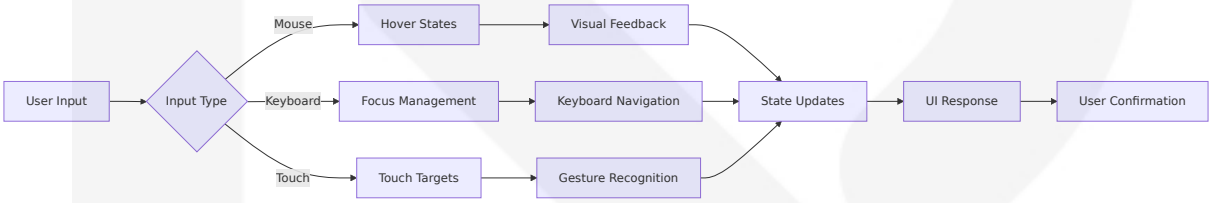
7.2.2 User Interaction Patterns

Navigation Patterns:

Interaction Type	Implementation	User Feedback	Accessibility
Primary Navigation	Sidebar with icons and labels	Active state highlighting	Keyboard navigation, ARIA labels
Content Browsing	Grid layout with hover states	Card elevation, preview on hover	Focus indicators, screen reader su

Interaction Type	Implementation	User Feedback	Accessibility
			pport
Search	Global search with autocomplete	Real-time suggestions, recent searches	Keyboard shortcuts, voice input support
Media Controls	Overlay controls with auto-hide	Visual feedback on interaction	Keyboard shortcuts, gesture support

Responsive Interaction Design:



7.2.3 State Management Patterns

UI State Categories:

State Type	Scope	Persistence	Synchronization
Application State	Global	Session storage	Cross-device sync
User Preferences	Global	Local storage	Cloud backup
Content State	Component-level	Memory	Real-time updates
Playback State	Media player	Local storage	Progress tracking

7.3 UI/BACKEND INTERACTION BOUNDARIES

7.3.1 Tauri IPC Communication

Command-Based Architecture:

Apps built with Tauri can ship with any number of pieces of an optional JS API and Rust API so that webviews can control the system via message passing. The frontend communicates with the Rust backend through Tauri's secure IPC system.

IPC Command Categories:

Command Category	Frontend Trigger	Backend Handler	Response Type
Content Discovery	Search input, category selection	TMDB API integration	JSON content data
User Management	Login, profile updates	Authentication service	User session data
Addon Operations	Install, configure, remove	Addon management system	Operation status
Media Playback	Play, pause, seek	Stream validation	Playback commands
Settings Management	Preference changes	Configuration service	Updated settings

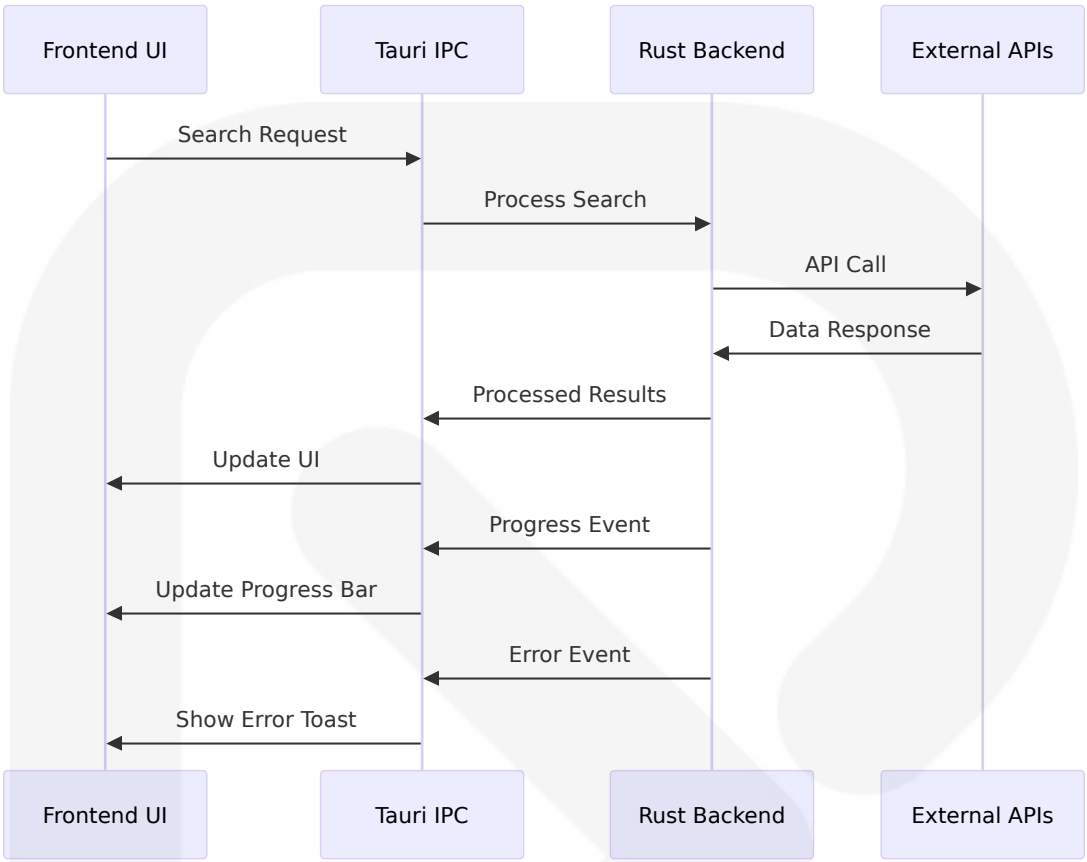
IPC Implementation Pattern:

```
// Frontend - Tauri API calls
import { invoke } from '@tauri-apps/api/core';

interface SearchParams {
  query: string;
  filters?: ContentFilters;
  page?: number;
}

interface ContentResult {
  id: string;
```

```
  title: string;
  overview: string;
  poster_path: string;
  release_date: string;
  vote_average: number;
}

// Search content
async function searchContent(params: SearchParams):
Promise<ContentResult[]> {
  try {
    const results = await invoke<ContentResult[]>('search_content',
params);
    return results;
  } catch (error) {
    console.error('Search failed:', error);
    throw error;
  }
}

// Install addon
async function installAddon(addonUrl: string): Promise<boolean> {
  try {
    const success = await invoke<boolean>('install_addon', { url:
addonUrl });
    return success;
  } catch (error) {
    console.error('Addon installation failed:', error);
    return false;
  }
}
```

7.3.2 Event-Driven Updates

Real-Time Communication:

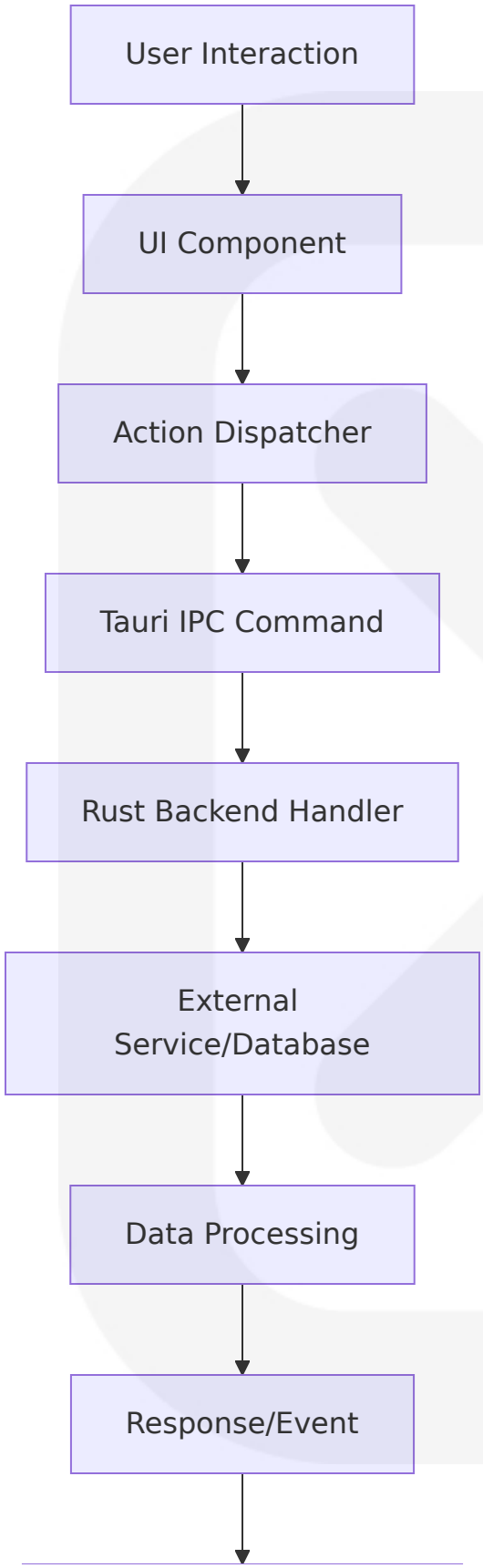


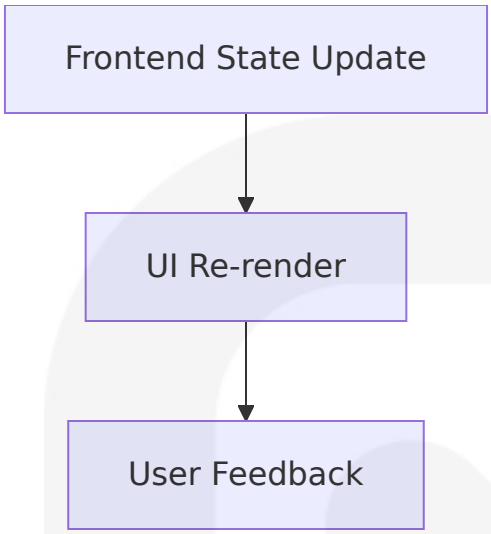
Event Subscription Pattern:

```
import { listen } from '@tauri-apps/api/event';

// Listen for download progress
await listen<ProgressPayload>('download-progress', (event) => {
  updateProgressBar(event.payload.percentage);
});

// Listen for addon status changes
await listen<AddonStatusPayload>('addon-status-changed', (event) => {
  updateAddonList(event.payload);
});

// Listen for playback events
await listen<PlaybackEventPayload>('playback-event', (event) => {
  handlePlaybackEvent(event.payload);
});
```

7.3.3 Data Flow Architecture

Unidirectional Data Flow:


User Interaction
UI Component
Action Dispatcher
Tauri IPC Command
Rust Backend Handler
External Service/Database
Data Processing
Response/Event



7.4 UI SCHEMAS

7.4.1 Component Data Schemas

Content Item Schema:

```
interface ContentItem {
  id: string;
  type: 'movie' | 'tv' | 'episode';
  title: string;
  overview: string;
  poster_path: string | null;
  backdrop_path: string | null;
  release_date: string;
  vote_average: number;
  vote_count: number;
  genres: Genre[];
  runtime?: number;
  status: 'released' | 'upcoming' | 'in_production';

  // TV-specific fields
  season_number?: number;
  episode_number?: number;
  episode_count?: number;

  // User-specific data
  is_favorite: boolean;
```

```
  is_in_watchlist: boolean;
  watch_progress?: WatchProgress;
}

interface Genre {
  id: number;
  name: string;
}

interface WatchProgress {
  current_time: number;
  total_time: number;
  percentage: number;
  last_watched: string;
  completed: boolean;
}
```

User Interface State Schema:

```
interface UIState {
  theme: 'light' | 'dark' | 'system';
  sidebar_collapsed: boolean;
  current_view: ViewType;
  search_query: string;
  selected_filters: ContentFilters;
  loading_states: LoadingStates;
  modal_stack: ModalState[];
  notifications: NotificationState[];
}

interface ContentFilters {
  genres: number[];
  year_range: [number, number];
  rating_range: [number, number];
  content_type: ('movie' | 'tv')[];
  sort_by: 'popularity' | 'rating' | 'release_date' | 'title';
  sort_order: 'asc' | 'desc';
}

interface LoadingStates {
  content_search: boolean;
  addon_installation: boolean;
```

```
  user_authentication: boolean;
  media_loading: boolean;
}
```

7.4.2 Form Validation Schemas

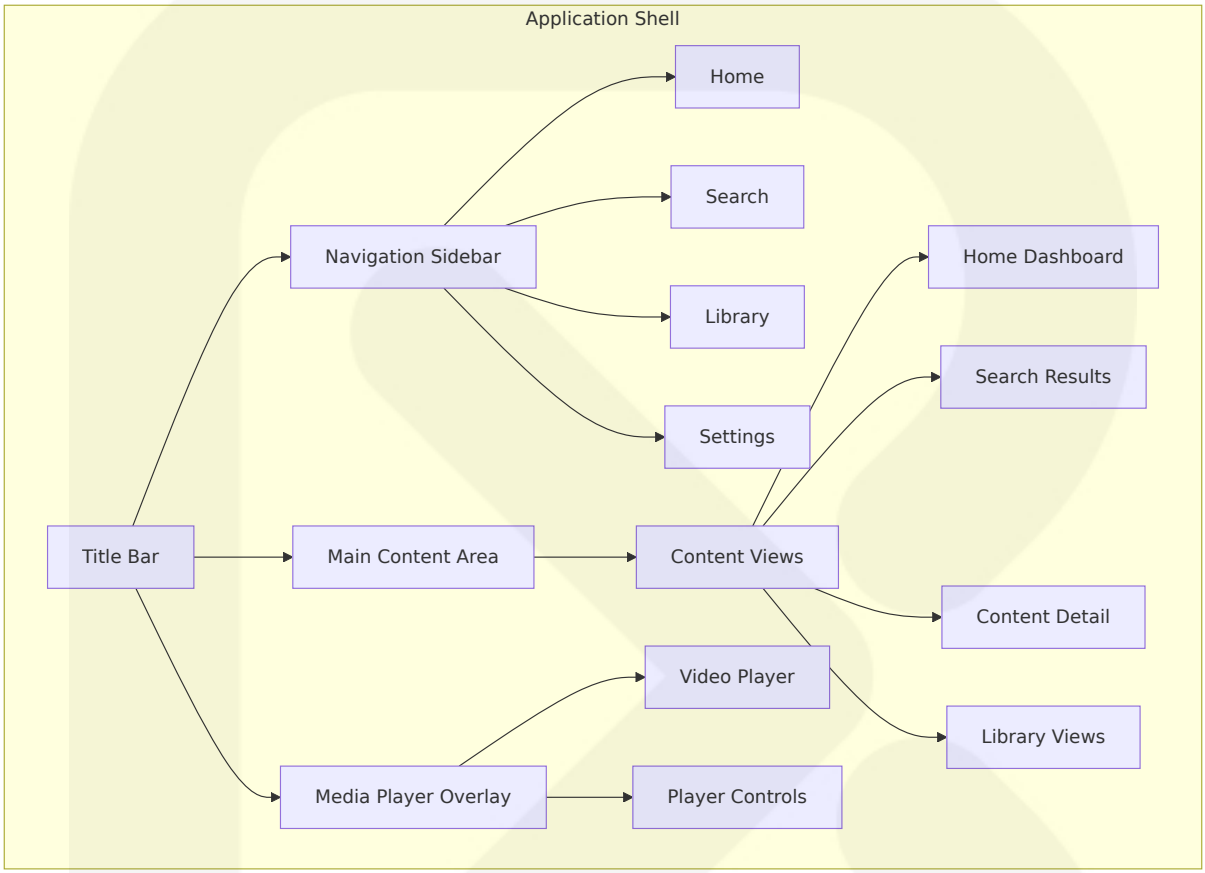
User Authentication Forms:

```
interface LoginFormData {
  email: string;
  password: string;
  remember_me: boolean;
}

interface RegistrationFormData {
  email: string;
  password: string;
  confirm_password: string;
  display_name: string;
  terms_accepted: boolean;
}

// Validation rules
const loginValidation = {
  email: {
    required: true,
    pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
    message: 'Please enter a valid email address'
  },
  password: {
    required: true,
    minLength: 8,
    message: 'Password must be at least 8 characters'
  }
};
```

7.5 SCREENS REQUIRED

7.5.1 Primary Application Screens

Main Application Layout:



7.5.2 Screen Specifications

1. Home Dashboard Screen

Purpose: Primary landing screen showcasing featured content, recommendations, and quick access to user library.

Layout Components:

- Hero banner with featured content
- Horizontal scrolling carousels for different content categories
- "Continue Watching" section
- Recently added content

- Trending/Popular content sections

Key Features:

- Personalized recommendations based on viewing history
- Quick play functionality
- Add to library actions
- Content preview on hover

2. Search Interface Screen

Purpose: Comprehensive search functionality with filtering and sorting capabilities.

Layout Components:

- Global search bar with autocomplete
- Filter sidebar (genre, year, rating, type)
- Search results grid with pagination
- Sort options (relevance, popularity, date, rating)
- Search history and suggestions

Key Features:

- Real-time search suggestions
- Advanced filtering options
- Search result previews
- Saved searches functionality

3. Content Detail Screen

Purpose: Detailed view of individual movies, TV shows, or episodes with comprehensive information and actions.

Layout Components:

- Large backdrop image with overlay information
- Content metadata (title, year, rating, genre, runtime)

- Synopsis and cast information
- Action buttons (Play, Add to Library, Share)
- Related content recommendations
- User reviews and ratings

Key Features:

- Trailer playback
- Season/episode navigation for TV shows
- Cast and crew information
- Similar content suggestions

4. Personal Library Screen

Purpose: User's personal collection management including watchlists, favorites, and viewing history.

Layout Components:

- Library navigation tabs (Watchlist, Favorites, History, Downloads)
- Content grid with sorting options
- Progress indicators for partially watched content
- Bulk actions for library management

Key Features:

- Multiple library categories
- Progress tracking visualization
- Bulk edit capabilities
- Export/import functionality

5. Video Player Screen

Purpose: Full-screen media playback with comprehensive controls and features.

Layout Components:

- Full-screen video display
- Overlay control bar (auto-hiding)
- Progress scrubber with thumbnail previews
- Volume control and quality selection
- Subtitle options and settings
- Casting controls

Key Features:

- Multiple quality options
- Subtitle support with customization
- Casting to external devices
- Keyboard shortcuts
- Picture-in-picture mode

6. Settings Screen

Purpose: Application configuration including user preferences, addon management, and system settings.

Layout Components:

- Settings navigation sidebar
- Tabbed content areas
- Form controls for preferences
- Addon management interface
- Account settings panel

Key Features:

- Theme selection (light/dark/auto)
- Playback preferences
- Addon installation and configuration
- Account management
- Data export/import options

7. Addon Management Screen

Purpose: Browse, install, and manage content source addons.

Layout Components:

- Available addons grid
- Installed addons list
- Addon detail panels
- Installation progress indicators
- Configuration interfaces

Key Features:

- Addon discovery and search
- One-click installation
- Addon configuration panels
- Health status monitoring
- Community ratings and reviews

7.5.3 Modal and Overlay Screens

Authentication Modals:

- Login dialog
- Registration form
- Password reset interface
- Account verification

Content Action Modals:

- Add to library dialog
- Share content interface
- Rating and review forms
- Report content dialog

System Modals:

- Settings quick access

- Notification center
- Download manager
- Error reporting interface

7.6 USER INTERACTIONS

7.6.1 Input Methods and Controls

Primary Interaction Methods:

Input Method	Implementation	Use Cases	Accessibility
Mouse/Trackpad	Standard pointer interactions	Navigation, content selection, media controls	Focus indicators, hover states
Keyboard	Comprehensive keyboard shortcuts	Navigation, search, media control	Full keyboard navigation, screen reader support
Touch	Touch-friendly targets (44px minimum)	Mobile/tablet interfaces	Gesture recognition, haptic feedback

Keyboard Shortcuts:

```
const keyboardShortcuts = {
  // Global navigation
  'Ctrl+1': 'Navigate to Home',
  'Ctrl+2': 'Navigate to Search',
  'Ctrl+3': 'Navigate to Library',
  'Ctrl+,': 'Open Settings',

  // Search
  'Ctrl+K': 'Focus search bar',
  'Escape': 'Clear search/Close modal',

  // Media playback
  'Space': 'Play/Pause',
```

```
    'ArrowLeft': 'Seek backward 10s',
    'ArrowRight': 'Seek forward 10s',
    'ArrowUp': 'Volume up',
    'ArrowDown': 'Volume down',
    'F': 'Toggle fullscreen',
    'M': 'Toggle mute',

    // Library management
    'Ctrl+D': 'Add to library',
    'Ctrl+F': 'Add to favorites',
    'Delete': 'Remove from library'
};
```

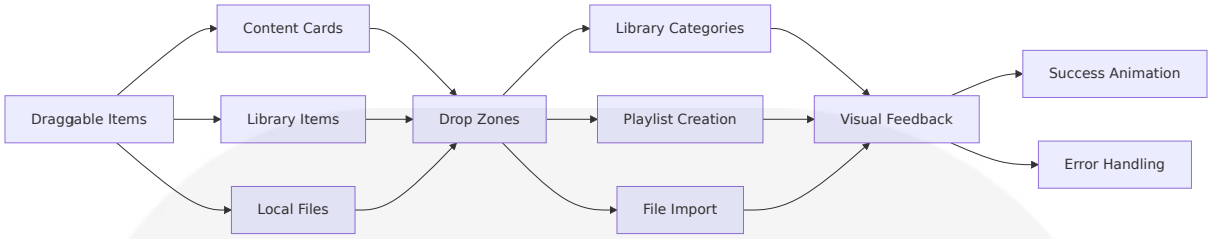
7.6.2 Gesture and Touch Interactions

Touch Gesture Support:

Gesture	Action	Context	Implementation
Tap	Select/Activate	All interactive elements	Standard touch events
Long Press	Context menu	Content items	500ms delay threshold
Swipe Left/Right	Navigate between screens	Content carousels	Horizontal scroll detection
Swipe Up/Down	Scroll content	Vertical lists	Vertical scroll with momentum
Pinch to Zoom	Zoom content	Image/video preview	Multi-touch scale detection
Double Tap	Quick action	Video player (play/pause)	300ms double-tap detection

7.6.3 Drag and Drop Functionality

Drag and Drop Use Cases:



Implementation Pattern:

```
// Drag and drop implementation
interface DragDropHandler {
  onDragStart: (item: ContentItem) => void;
  onDragOver: (event: DragEvent) => void;
  onDrop: (item: ContentItem, target: DropTarget) => void;
  onDragEnd: () => void;
}

const dragDropConfig = {
  dragTypes: ['content-item', 'library-item', 'local-file'],
  dropZones: ['watchlist', 'favorites', 'custom-playlist'],
  visualFeedback: {
    dragCursor: 'grabbing',
    dropZoneHighlight: 'border-primary',
    invalidDrop: 'border-error'
  }
};
```

7.6.4 Context Menus and Quick Actions

Context Menu System:

Context	Menu Items	Actions
Content Item	Play, Add to Library, Share, Details, Remove	Direct content actions
Library Item	Play, Remove, Move to Category, Mark as Watched	Library management
Search Result	Play, Add to Library, View Details, Similar Content	Discovery actions

Context	Menu Items	Actions
Video Player	Quality Settings, Subtitles, Audio Tracks, Cast	Playback controls

7.7 VISUAL DESIGN CONSIDERATIONS

7.7.1 Design System and Brand Identity

Visual Hierarchy:

The interface follows a clear visual hierarchy inspired by modern streaming platforms, with emphasis on content discovery and ease of navigation. Stremio provides an unparalleled viewing experience for movies, TV shows, web videos, live TV, and more with its sleek interface, broad content library, and powerful add-on capabilities.

Color Palette:

Color Role	Dark Theme	Light Theme	Usage
Primary	#8b5cf6 (Purple)	#7c3aed (Purple)	Brand elements, CTAs, active states
Secondary	#f59e0b (Amber)	#d97706 (Amber)	Accent elements, ratings, highlights
Background	#0a0a0a (Near Black)	#ffffff (White)	Main background
Surface	#1a1a1a (Dark Gray)	#f8fafc (Light Gray)	Cards, panels, elevated surfaces
Text Primary	#ffffff (White)	#0f172a (Dark Blue)	Primary text content
Text Secondary	#a1a1aa (Gray)	#475569 (Blue Gray)	Secondary text, metadata

7.7.2 Typography System

Font Hierarchy:

```
/* Typography Scale */
.text-display {
  font-size: 2.25rem; /* 36px */
  font-weight: 700;
  line-height: 1.2;
  letter-spacing: -0.025em;
}

.text-heading-1 {
  font-size: 1.875rem; /* 30px */
  font-weight: 600;
  line-height: 1.3;
}

.text-heading-2 {
  font-size: 1.5rem; /* 24px */
  font-weight: 600;
  line-height: 1.4;
}

.text-heading-3 {
  font-size: 1.25rem; /* 20px */
  font-weight: 500;
  line-height: 1.4;
}

.text-body {
  font-size: 1rem; /* 16px */
  font-weight: 400;
  line-height: 1.6;
}

.text-caption {
  font-size: 0.875rem; /* 14px */
  font-weight: 400;
  line-height: 1.5;
  color: var(--color-text-secondary);
}
```

7.7.3 Layout and Grid System

Responsive Grid Architecture:

Grid Specifications:

Breakpoint	Container Width	Columns	Gutter	Margins
Mobile	100%	2-3	16px	16px
Tablet	100%	4-5	20px	24px
Desktop	1200px max	6-8	24px	32px
Large Desktop	1400px max	8-10	32px	48px

7.7.4 Animation and Micro-interactions

Animation Principles:

Animation Type	Duration	Easing	Purpose
Page Transitions	300ms	ease-out	Smooth navigation
Hover Effects	150ms	ease-in-out	Interactive feedback
Loading States	Continuous	linear	Progress indication
Modal Animations	250ms	ease-out	Focus management
Micro-interactions	100ms	ease-out	Immediate feedback

Key Animation Patterns:

```
/* Smooth transitions for interactive elements */
.interactive-element {
```

```
  transition: all 0.15s ease-in-out;
}

.interactive-element:hover {
  transform: translateY(-2px);
  box-shadow: 0 8px 25px rgba(0, 0, 0, 0.15);
}

/* Loading animation */
@keyframes pulse {
  0%, 100% { opacity: 1; }
  50% { opacity: 0.5; }
}

.loading-skeleton {
  animation: pulse 2s cubic-bezier(0.4, 0, 0.6, 1) infinite;
}

/* Page transition */
.page-enter {
  opacity: 0;
  transform: translateX(20px);
}

.page-enter-active {
  opacity: 1;
  transform: translateX(0);
  transition: opacity 300ms ease-out, transform 300ms ease-out;
}
```

7.7.5 Accessibility and Inclusive Design

Accessibility Standards:

The interface adheres to WCAG 2.1 AA standards with the following implementations:

Accessibility Feature	Implementation	Benefit
Color Contrast	4.5:1 minimum ratio	Improved readability for all users
Keyboard Navigation	Full keyboard accessibility	Users with motor disabilities
Screen Reader Support	ARIA labels and landmarks	Users with visual impairments
Focus Management	Visible focus indicators	Clear navigation feedback
Alternative Text	Descriptive alt text for images	Content accessibility
Reduced Motion	Respects prefers-reduced-motion	Users sensitive to motion

Inclusive Design Features:

```
/* Respect user motion preferences */
@media (prefers-reduced-motion: reduce) {
  * {
    animation-duration: 0.01ms !important;
    animation-iteration-count: 1 !important;
    transition-duration: 0.01ms !important;
  }
}

/* High contrast mode support */
@media (prefers-contrast: high) {
  :root {
    --color-border: #000000;
    --color-text-primary: #000000;
    --color-background: #ffffff;
  }
}

/* Focus indicators */
.focusable:focus-visible {
  outline: 2px solid var(--color-primary);
```

```
  outline-offset: 2px;
}
```

7.7.6 Dark/Light Theme Implementation

Theme Toggle System:

The application supports automatic theme detection based on system preferences, with manual override options.

```
interface ThemeConfig {
  mode: 'light' | 'dark' | 'system';
  customColors?: Partial<ColorPalette>;
  animations: boolean;
  reducedMotion: boolean;
}

const themeManager = {
  detectSystemTheme(): 'light' | 'dark' {
    return window.matchMedia('(prefers-color-scheme: dark)').matches ?
'dark' : 'light';
  },

  applyTheme(theme: 'light' | 'dark') {
    document.documentElement.setAttribute('data-theme', theme);
    // Update meta theme-color for mobile browsers
    const metaTheme = document.querySelector('meta[name="theme-
color"]');
    if (metaTheme) {
      metaTheme.setAttribute('content', theme === 'dark' ? '#0a0a0a' :
'#ffffff');
    }
  },

  toggleTheme() {
    const current = document.documentElement.getAttribute('data-
theme');
    const next = current === 'dark' ? 'light' : 'dark';
    this.applyTheme(next);
    localStorage.setItem('theme-preference', next);
```

```
  }
};
```

This comprehensive UI design framework ensures the Stremio Clone Desktop Application delivers a modern, accessible, and performant user experience that rivals commercial streaming platforms while maintaining the flexibility and extensibility required for a community-driven media center application.

8. Infrastructure

8.1 DEPLOYMENT ENVIRONMENT

8.1.1 Target Environment Assessment

Detailed Infrastructure Architecture is not applicable for this system as the Stremio Clone Desktop Application is a standalone desktop application built with Tauri 2.0. Tauri is a framework for building tiny and fast binaries for all major desktop (macOS, linux, windows) and mobile (iOS, Android) platforms. Developers can integrate any frontend framework that compiles to HTML, JavaScript, and CSS for building their user experience while leveraging languages such as Rust, Swift, and Kotlin for backend logic when needed.

Unlike web applications or distributed systems that require server infrastructure, cloud services, or container orchestration, this desktop application runs entirely on the user's local machine. In a Tauri application the frontend is written in your favorite web frontend stack. This runs inside the operating system WebView and communicates with the application core written mostly in Rust.

Environment Type Classification:

Environment Aspect	Classification	Justification
Deployment Model	Local Installation	Application runs entirely on user's desktop
Infrastructure Requirements	None	No servers, databases, or cloud services required
Geographic Distribution	User-distributed	Each user downloads and installs locally
Resource Requirements	Client-side only	Uses local system resources (CPU, memory, storage)

8.1.2 Local System Requirements

Platform Support Matrix:

Platform	Minimum Requirements	Recommended Requirements	WebView Engine
Windows	Windows 10 (1803+), WebView2 Runtime	Windows 11, 8GB RAM, SSD	WebView2 on Windows
macOS	macOS 10.15+, Xcode Command Line Tools	macOS 12+, 8GB RAM, SSD	WKWebView on macOS
Linux	webkit2gtk 4.1 for Tauri v2 (for example Ubuntu 22.04)	Ubuntu 22.04+, 8GB RAM, SSD	WebKitGTK on Linux

Development Environment Requirements:

Component	Purpose	Installation Method
Rust Toolchain	Tauri is built with Rust and requires it for development	rustup installer
Node.js	Frontend build tools	Official installer (v18+)

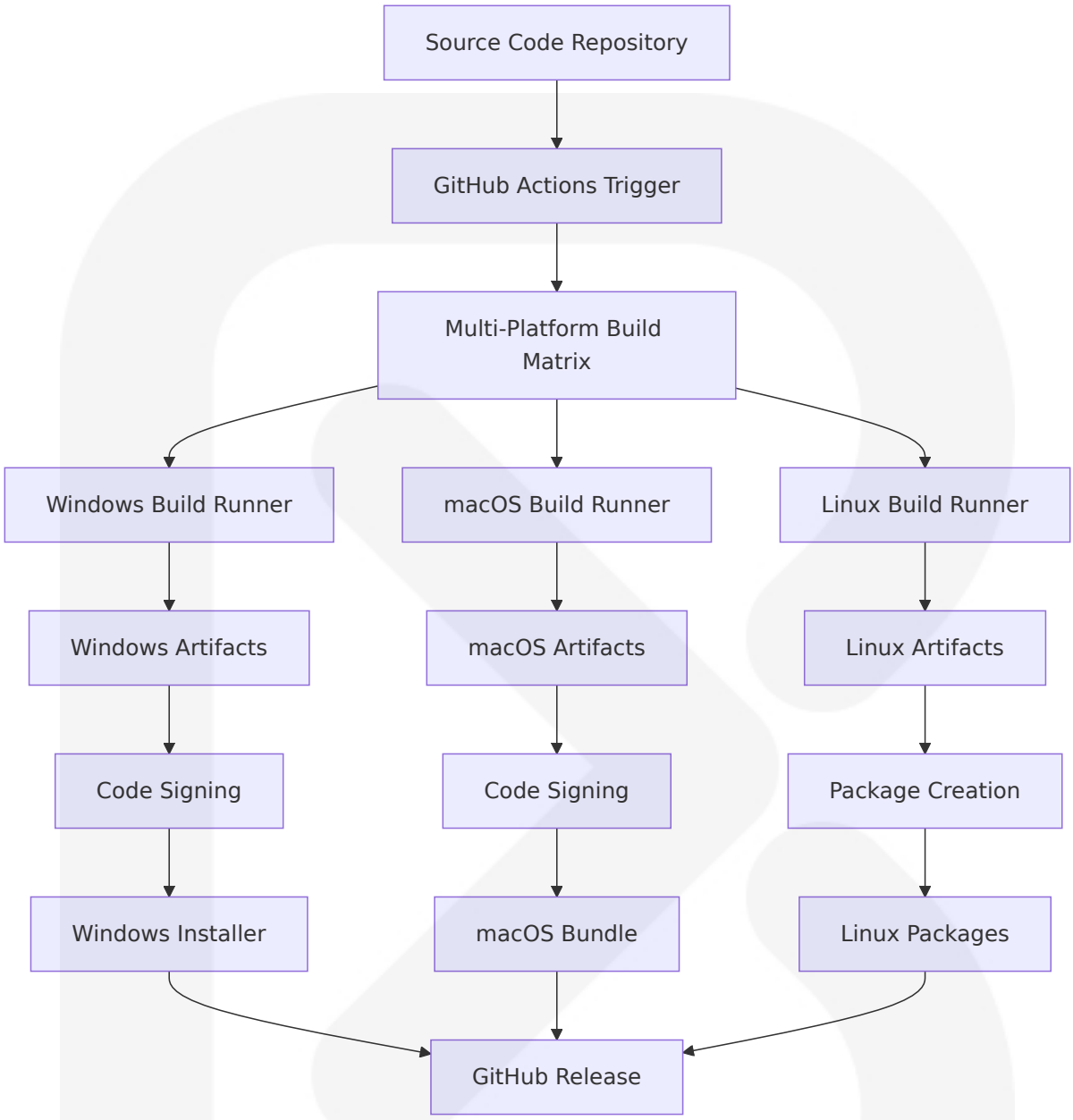
Component	Purpose	Installation Method
System Dependencies	Platform-specific build tools	Package managers

8.2 BUILD AND DISTRIBUTION REQUIREMENTS

8.2.1 Build Pipeline Architecture

Cross-Platform Build Strategy:

Tauri relies heavily on native libraries and toolchains, so meaningful cross-compilation is not possible at the current moment. The next best option is to compile utilizing a CI/CD pipeline hosted on something like GitHub Actions, Azure Pipelines, GitLab, or other options. The pipeline can run the compilation for each platform simultaneously making the compilation and release process much easier.



8.2.2 CI/CD Pipeline Implementation

GitHub Actions Workflow:

This GitHub Action has three main usages: test the build pipeline of your Tauri app, uploading Tauri artifacts to an existing release, and creating a new release with the Tauri artifacts. This example shows the most common use case for tauri-action. The action will build the app, create a GitHub

release itself, and upload the app bundles to the newly created release. This is generally the simplest way to release your Tauri app.

Build Matrix Configuration:

Platform	Runner	Build Time	Artifact Types
Windows	`windows-latest`	15-20 minutes	`.exe` (NSIS), `.msi` (WiX)
macOS	`macos-latest`	20-25 minutes	`.app`, `.dmg`
Linux	`ubuntu-22.04`	10-15 minutes	`.deb`, `.rpm`, `.AppImage`

Example CI/CD Workflow:

```
name: 'Build and Release'
on:
  push:
    branches: [release]
  pull_request:
    branches: [main]

jobs:
  build:
    permissions:
      contents: write
    strategy:
      fail-fast: false
      matrix:
        platform: [macos-latest, ubuntu-22.04, windows-latest]
    runs-on: ${{ matrix.platform }}

    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'
```

```
      - name: Setup Rust
        uses: dtolnay/rust-toolchain@stable
        with:
          targets: ${{ matrix.platform == 'macos-latest' && 'aarch64-
apple-darwin,x86_64-apple-darwin' || '' }}

      - name: Rust cache
        uses: swatinem/rust-cache@v2
        with:
          workspaces: './src-tauri -> target'

      - name: Install Linux dependencies
        if: matrix.platform == 'ubuntu-22.04'
        run: |
          sudo apt-get update
          sudo apt-get install -y libwebkit2gtk-4.1-dev
libappindicator3-dev librsvg2-dev patchelf

      - name: Install frontend dependencies
        run: npm ci

      - name: Build and release
        uses: tauri-apps/tauri-action@v0
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          tagName: v__VERSION__
          releaseName: 'Stremio Clone v__VERSION__'
          releaseBody: 'See the assets to download and install this
version.'
          releaseDraft: true
          prerelease: false
```

8.2.3 Distribution Formats

Platform-Specific Bundle Formats:

Built-in app bundler to create app bundles in formats like .app, .dmg, .deb, .rpm, .AppImage and Windows installers like .exe (via NSIS) and .msi (via WiX).

Platform	Primary Format	Alternative Formats	Distribution Method
Windows	NSIS Installer (`.exe`)	MSI Installer (`.msi`)	Direct download, Microsoft Store (future)
macOS	DMG Bundle (`.dmg`)	App Bundle (`.app`)	Direct download, App Store (future)
Linux	AppImage (`.AppImage`)	DEB (`.deb`), RPM (`.rpm`)	Direct download, package repositories

8.2.4 Code Signing Requirements

Platform-Specific Signing:

Code signing enhances the security of your application by applying a digital signature to your application's executables and bundles, validating your identity of the provider of your application. Signing is required on most platforms. See the documentation for each platform for more information.

Platform	Signing Requirement	Certificate Type	Implementation
Windows	Recommended	Code Signing Certificate	Automated via GitHub Actions
macOS	Both methods requires code signing, and distributing outside the App Store also requires notarization	Apple Developer Certificate	Automated signing and notarization
Linux	Optional	GPG Signing	Package-specific signing

8.2.5 Version Management

Versioning Strategy:

Your application version can be defined in the tauri.conf.json > version configuration option, which is the recommended way for managing the app version. If that config value is not set, Tauri uses the package > version value from your src-tauri/Cargo.toml file instead.

Version Component	Source	Format	Example
Application Version	`tauri.conf.json`	Semantic Versioning	`2.1.0`
Build Number	CI/CD Pipeline	Incremental	`build.123`
Git Tag	Repository Tags	`v{version}`	`v2.1.0`

8.3 DEVELOPMENT INFRASTRUCTURE

8.3.1 Development Environment Setup

Local Development Requirements:

The first time you run this command, the Rust package manager may need several minutes to download and build all the required packages. Since they are cached, subsequent builds are much faster, as only your code needs rebuilding. Once Rust has finished building, the webview opens, displaying your web app.

Component	Installation	Purpose	Platform Notes
Rust Toolchain	`rustup` installer	Backend compilation	For full support for Tauri and tools like trunk make sure the MSVC Rust toolchain is the selected default host triple in the installer dialog. Depending on your system it should be either x86_64-pc-windows-msvc, i686-pc-windo

Component	Installation	Purpose	Platform Notes
			ws-msvc, or aarch64-pc-windows-msvc
Node.js	Official installer	Frontend build tools	Version 18+ recommended
Tauri CLI	`cargo install tauri-cli`	Development commands	Global installation

Platform-Specific Dependencies:

Platform	Required Dependencies	Installation Command
Windows	Microsoft C++ Build Tools, Microsoft Edge WebView2	Visual Studio Installer
macOS	Xcode and various macOS development dependencies	App Store or Apple Developer
Linux	Various system dependencies for development on Linux. These may be different depending on your distribution	Package manager (apt, yum, etc.)

8.3.2 Development Workflow

Development Commands:

```
# Start development server
tauri dev

#### Build for production
tauri build

#### Run tests
cargo test
```

```
#### Frontend development (parallel)
npm run dev
```

Hot Reload and Debugging:

You can make changes to your web app, and if your tooling supports it, the webview should update automatically, just like a browser. You can open the Web Inspector to debug your application by performing a right-click on the webview and clicking "Inspect" or using the Ctrl + Shift + I shortcut on Windows and Linux or Cmd + Option + I shortcut on macOS.

8.4 MONITORING AND MAINTENANCE

8.4.1 Application Monitoring

Client-Side Monitoring Strategy:

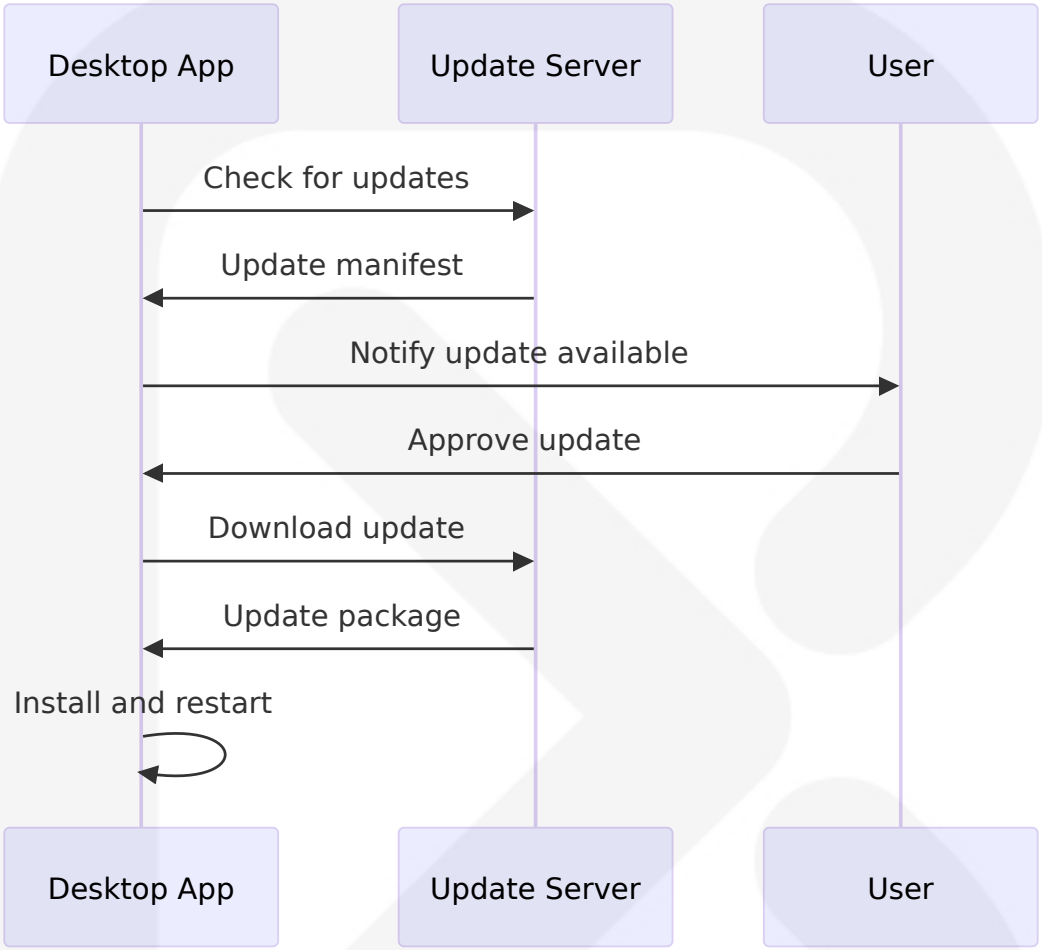
Since this is a desktop application, traditional server monitoring is not applicable. Instead, monitoring focuses on:

Monitoring Type	Implementation	Data Collection	User Benefit
Crash Reporting	Built-in error handling	Local crash logs	Improved stability
Performance Metrics	Application telemetry	Response times, memory usage	Better performance
Usage Analytics	Optional user consent	Feature usage patterns	Enhanced user experience
Update Monitoring	Version checking	Update success rates	Reliable updates

8.4.2 Update Distribution

Automatic Update System:

Tauri provides built-in support for application updates through the updater plugin, enabling seamless distribution of new versions to users.



8.4.3 Maintenance Procedures

Maintenance Categories:

Maintenance Type	Frequency	Automation Level	Responsibility
Dependency Updates	Monthly	Semi-automated	Development team
Security Patches	As needed	Manual review	Security team
Feature Updates	Quarterly	Manual	Product team

Maintenance Type	Frequency	Automation Level	Responsibility
Bug Fixes	As needed	Automated testing	Development team

8.5 COST CONSIDERATIONS

8.5.1 Development Costs

Infrastructure Cost Analysis:

Cost Category	Monthly Cost	Annual Cost	Notes
GitHub Actions	$0-50	$0-600	Free tier: 2,000 minutes/month
Code Signing Certificates	N/A	$200-500	One-time annual cost
Development Tools	$0	$0	Open source toolchain
Distribution	$0	$0	Direct download, no hosting costs

8.5.2 Operational Costs

Ongoing Operational Expenses:

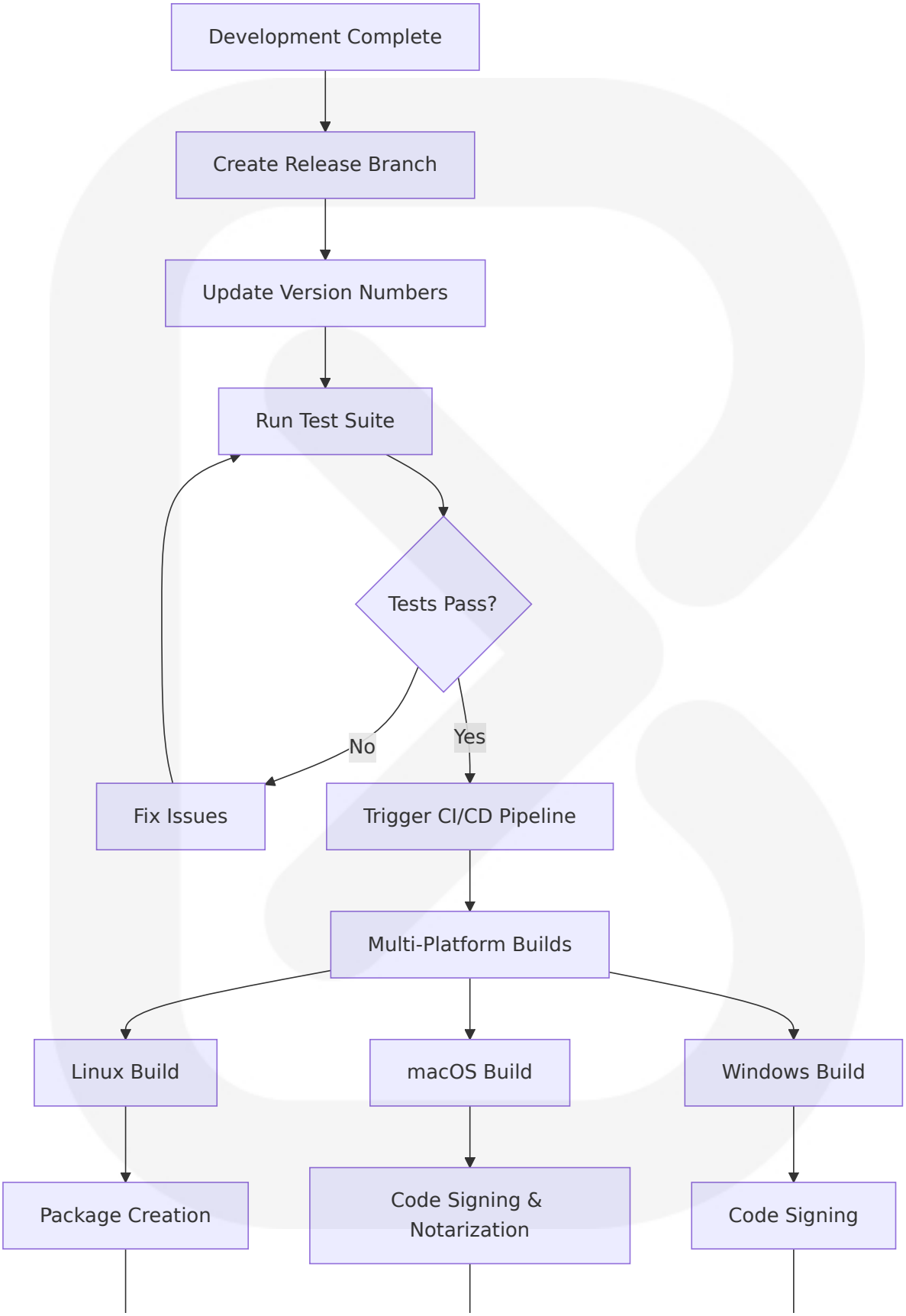
Expense Type	Cost Range	Frequency	Justification
Certificate Renewal	$200-500	Annual	Code signing requirements
CI/CD Minutes	$0-100	Monthly	Build automation
Storage	$0-20	Monthly	Artifact storage

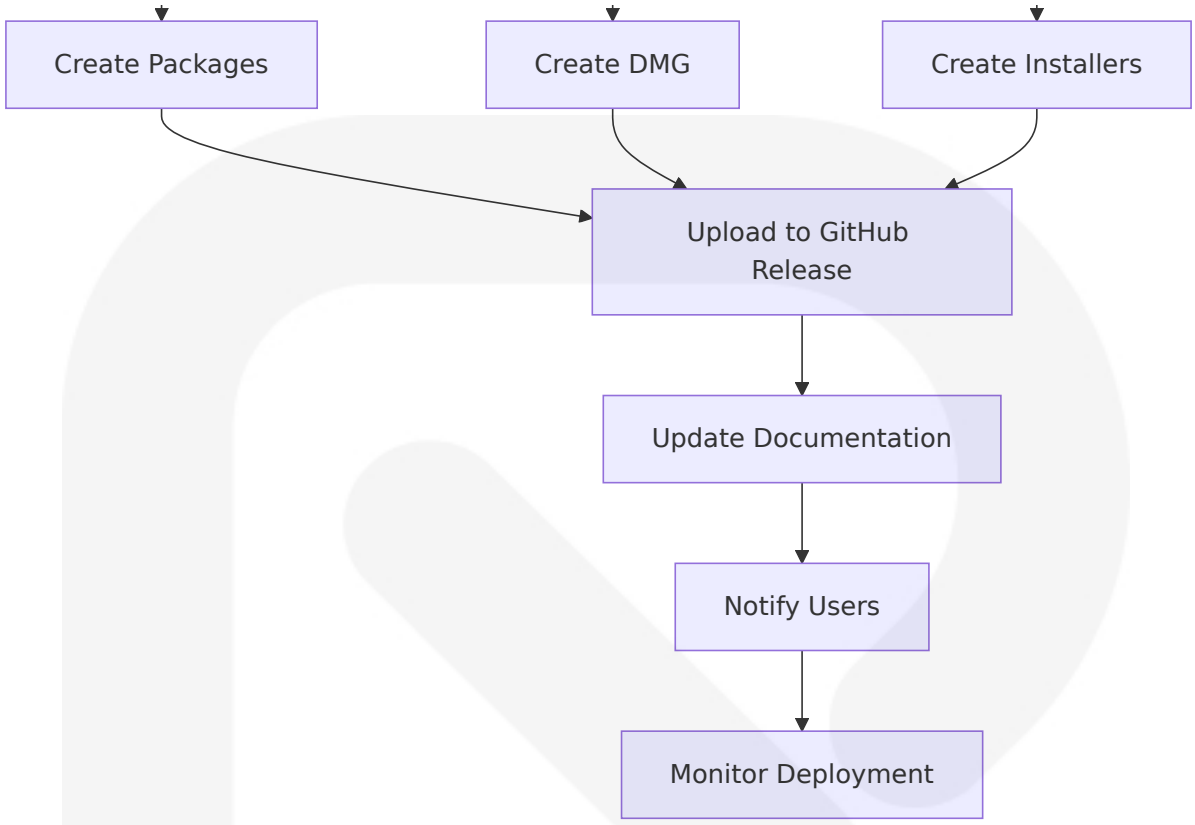
Expense Type	Cost Range	Frequency	Justification
Bandwidth	$0	N/A	User downloads from GitHub

8.6 DEPLOYMENT WORKFLOW

8.6.1 Release Process

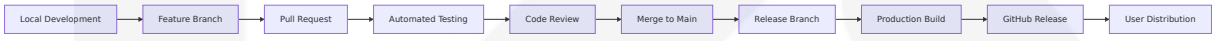
Deployment Workflow Diagram:


Development Complete
Create Release Branch
Update Version Numbers
Run Test Suite
Tests Pass?
No
Yes
Fix Issues
Trigger CI/CD Pipeline
Multi-Platform Builds
Linux Build
macOS Build
Windows Build
Package Creation
Code Signing & Notarization
Code Signing



8.6.2 Environment Promotion Flow

Development to Production Pipeline:



8.7 BACKUP AND RECOVERY

8.7.1 Source Code Protection

Repository Backup Strategy:

Backup Type	Frequency	Storage Location	Retention Period
Git Repository	Real-time	GitHub (primary)	Indefinite
Repository Mirror	Daily	GitLab (secondary)	Indefinite

Backup Type	Frequency	Storage Location	Retention Period
Release Artifacts	Per release	GitHub Releases	2 years
Build Artifacts	Per build	GitHub Actions	90 days

8.7.2 User Data Considerations

Local Data Management:

Since the application stores user data locally, backup and recovery focuses on:

- **User Education**: Documentation on backing up personal libraries and settings
- **Export Functionality**: Built-in data export features for user migration
- **Import Capabilities**: Ability to restore user data from backups
- **Sync Options**: Optional cloud synchronization for user preferences

8.8 SECURITY CONSIDERATIONS

8.8.1 Build Security

Secure Build Pipeline:

Security Measure	Implementation	Purpose
Dependency Scanning	`cargo audit` in CI/CD	Vulnerability detection
Code Signing	Platform-specific certificates	Authenticity verification
Artifact Verification	Checksums and signatures	Integrity validation

Security Measure	Implementation	Purpose
Secure Secrets	GitHub Secrets management	Certificate protection

8.8.2 Distribution Security

Secure Distribution Practices:

- **HTTPS Downloads**: All downloads served over encrypted connections
- **Signature Verification**: Digital signatures for all distributed binaries
- **Update Security**: Signed updates with rollback capabilities
- **User Verification**: Clear instructions for verifying download authenticity

This infrastructure approach ensures that the Stremio Clone Desktop Application can be efficiently developed, built, tested, and distributed across all target platforms while maintaining security, reliability, and cost-effectiveness. The focus on automation and standardized tooling minimizes operational overhead while maximizing development productivity.

9. Appendices

9.1 ADDITIONAL TECHNICAL INFORMATION

9.1.1 Tauri Development Environment SetupDetailed Development Environment Setup:

Platform	Prerequisites	Installation Commands
Windows	Microsoft C++ Build Tools, Microsoft Edge WebView2 (pre-installed on Windows 10 1803+), MSVC Rust toolchain	`rustup toolchain install stable-x86_64-pc-windows-msvc`
macOS	Xcode from App Store or Apple Developer website, launch Xcode after installing	`xcode-select --install` (for desktop-only)
Linux	webkit2gtk 4.1 for Tauri v2 (Ubuntu 22.04+), various system dependencies	`sudo apt-get install libwebkit2gtk-4.1-dev`

Project Structure and Configuration:Typical Tauri Project Structure:

```
stremio-clone/
├── src/                          # Frontend source code
│   ├── index.html                # Main HTML entry point
│   ├── main.js                   # Frontend JavaScript/TypeScript
│   ├── styles.css                # CSS styling
│   └── components/               # UI components (if using framework)
├── src-tauri/                    # Rust backend directory
│   ├── Cargo.toml                # Rust dependencies and metadata
│   ├── tauri.conf.json           # Main Tauri configuration
│   ├── build.rs                  # Build script
│   ├── icons/                    # Application icons
│   └── src/
│       ├── main.rs               # Entry point for desktop
│       └── lib.rs                # Shared library code
├── package.json                  # Frontend dependencies (if using Node.js)
└── README.md
```

9.1.2 Rust Crates and DependenciesEssential Rust Crates for Stremio Clone:

Category	Crate	Version	Purpose
Core Framework	`tauri`	2.0+	Desktop application framework
HTTP Client	`reqwest`	0.12+	TMDB API and addon communication
Serialization	`serde`	1.0+	JSON handling for APIs and configuration
Database	`sqlx`	0.7+	SQLite database operations with async support

Additional Recommended Crates:

Category	Crate	Purpose	Usage in Project
Async Runtime	`tokio`	Asynchronous operations	Background tasks, HTTP requests
Error Handling	`anyhow`	Error propagation	Simplified error handling
UUID Generation	`uuid`	Unique identifiers	Content and session IDs
Date/Time	`chrono`	Date/time operations	Timestamps, scheduling

9.1.3 Frontend Framework Integration

Supported Frontend Frameworks:

create-tauri-app currently includes templates for vanilla (HTML, CSS and JavaScript without a framework), Vue.js, Svelte, React, SolidJS, Angular, Preact, Yew, Leptos, and Sycamore.

Framework	Bundle Size	Performance	Learning Curve	Recommendation for Media App
Svelte	Smallest	Excellent	Moderate	□□□ Recommended for performan

Framework	Bundle Size	Performance	Learning Curve	Recommendation for Media App
				ce
React	Large	Good	Moderate	□□ Good for complex state management
Vue.js	Medium	Good	Easy	□□ Good for rapid development
Vanilla JS	Minimal	Excellent	Easy	□□□ Best for maximum control

9.1.4 Video Player Integration Options

Web-Based Video Players:

Player	Features	License	Integration Complexity
Video.js	HLS, DASH, plugins, themes	Apache 2.0	Low
Plyr	Modern UI, accessibility	MIT	Low
JW Player	Advanced features, analytics	Commercial	Medium
Custom HTML5	Basic playback, full control	N/A	High

9.1.5 Security Considerations for Desktop Applications

Tauri Security Model:

A grouping and boundary mechanism developers can use to isolate access to the IPC layer. It controls application windows' and webviews' fine grained

access to the Tauri core, application, or plugin commands. If a webview or its window is not matching any capability then it has no access to the IPC layer at all.

Security Best Practices:

Security Layer	Implementation	Benefit
Capability System	Granular permission control	Minimize attack surface
Input Validation	Sanitize all user inputs	Prevent injection attacks
HTTPS Enforcement	Secure addon communication	Protect data in transit
Content Security Policy	Restrict resource loading	Prevent XSS attacks

9.2 GLOSSARY

A

Addon: A remote plugin that extends the functionality of the Stremio Clone by providing additional content sources or features. Unlike traditional plugins, addons run on remote servers rather than locally.

API Key: A unique identifier used to authenticate requests to external services like TMDB (The Movie Database).

B

Bundle: The packaged application ready for distribution, containing all necessary files and dependencies for a specific platform.

C

CORS (Cross-Origin Resource Sharing): A security feature that allows or restricts web pages to access resources from other domains. Required

for addon communication.

Content Metadata: Detailed information about movies, TV shows, or other media content, including titles, descriptions, cast, ratings, and images.

D

Desktop Application: A software application designed to run on desktop operating systems (Windows, macOS, Linux) rather than in a web browser.

E

Electron Alternative: Tauri serves as a lightweight alternative to Electron for building cross-platform desktop applications.

F

Frontend Framework: A software framework designed to support the development of user interfaces, such as React, Vue.js, or Svelte.

H

Hot Module Replacement (HMR): A development feature that allows updating modules in a running application without requiring a full restart.

I

IPC (Inter-Process Communication): The mechanism by which the frontend (WebView) and backend (Rust) processes communicate securely in Tauri applications.

J

JSONB: A binary representation of JSON data in SQLite that provides faster processing and smaller storage compared to text JSON.

M

Manifest: A JSON file that describes an addon's capabilities, endpoints, and metadata required for integration with the Stremio Clone.

Media Center: A software application designed to organize, manage, and play digital media content from various sources.

P

Plugin System: An architecture that allows extending application functionality through modular components that can be added or removed.

R

Rate Limiting: A technique to control the number of requests made to an API within a specific time period to prevent abuse and ensure fair usage.

Rust Crate: A package of Rust code that can be shared and reused across different projects, similar to libraries in other programming languages.

S

SQLite: A lightweight, embedded relational database engine that stores data in a single file, ideal for desktop applications.

Streaming: The process of delivering media content over a network in a continuous flow, allowing playback to begin before the entire file is downloaded.

T

Tauri: A framework for building lightweight, secure desktop applications using web technologies for the frontend and Rust for the backend.

TMDB (The Movie Database): A community-built movie and TV database that provides comprehensive metadata and images for media content.

W

WebView: A system component that renders web content within native applications, providing the display engine for Tauri's frontend.

Watchlist: A user-curated list of movies, TV shows, or other content they intend to watch in the future.

9.3 ACRONYMS

Acronym	Full Form	Context
API	Application Programming Interface	External service integration
CORS	Cross-Origin Resource Sharing	Web security mechanism
CPU	Central Processing Unit	System performance
CSS	Cascading Style Sheets	Frontend styling
DASH	Dynamic Adaptive Streaming over HTTP	Video streaming protocol
DMG	Disk Image	macOS application package format
DNS	Domain Name System	Network name resolution
GDPR	General Data Protection Regulation	Privacy compliance
GPU	Graphics Processing Unit	Hardware acceleration
HLS	HTTP Live Streaming	Video streaming protocol
HMR	Hot Module Replacement	Development feature
HTML	HyperText Markup Language	Web content structure
HTTP	HyperText Transfer Protocol	Web communication
HTTPS	HyperText Transfer Protocol Secure	Secure web communication

Acronym	Full Form	Context
IDE	Integrated Development Environment	Development tools
IPC	Inter-Process Communication	Process communication
JSON	JavaScript Object Notation	Data interchange format
JSONB	JSON Binary	Binary JSON format in SQLite
JWT	JSON Web Token	Authentication token format
LRU	Least Recently Used	Cache eviction algorithm
MSI	Microsoft Installer	Windows package format
MSRV	Minimum Supported Rust Version	Rust compatibility
NSIS	Nullsoft Scriptable Install System	Windows installer
OS	Operating System	Platform environment
RAM	Random Access Memory	System memory
REST	Representational State Transfer	API architecture
RPM	Red Hat Package Manager	Linux package format
SDK	Software Development Kit	Development tools
SLA	Service Level Agreement	Service guarantees
SQL	Structured Query Language	Database query language
SRT	SubRip Text	Subtitle file format
SSL	Secure Sockets Layer	Security protocol
TLS	Transport Layer Security	Security protocol
TMDB	The Movie Database	Content metadata service
TTL	Time To Live	Cache expiration

Acronym	Full Form	Context
UI	User Interface	Application interface
URL	Uniform Resource Locator	Web address
UX	User Experience	User interaction design
VTT	WebVTT	Web video subtitle format
WAL	Write-Ahead Logging	SQLite journal mode
WCAG	Web Content Accessibility Guidelines	Accessibility standards
WebRTC	Web Real-Time Communication	Peer-to-peer communication
WRY	WebView Rendering librarY	Tauri's WebView abstraction
XSS	Cross-Site Scripting	Security vulnerability