- Agentic Neurodata Conversion System - Requirements Specification
    - Executive Summary
        - Complete Feature Set
        - Scope Constraints
    - System Architecture Overview
    - Personas & Stakeholders
        - Persona 1: User (Data Scientist / Researcher)
        - Persona 2: System (Technical Requirements)
        - Persona 3: Developer/Maintainer
    - User Stories by Epic
    - Epic 1: MCP Server Infrastructure
        - Story 1.1: MCP Server Foundation
        - Story 1.2: Message Routing System
        - Story 1.3: Context Management
    - Epic 2: Global State Management (Single Session)
        - Story 2.1: Global State Object
        - Story 2.2: Stage Tracking
    - Epic 3: LLM Service Foundation
        - Story 3.1: LLM Service Abstract Interface
        - Story 3.2: Anthropic Claude Integration
    - Epic 4: Conversation Agent - User Interaction
        - Story 4.1: Conversation Agent Foundation
        - Story 4.2: Initial Metadata Validation Handler
        - Story 4.3: Improvement Approval Handler (Deprecated - See Stories 8.2, 8.3, 8.3a)
        - Story 4.4: Correction Context Analysis with LLM
        - Story 4.5: User Input Request Generator with LLM
        - Story 4.6: User Input Validation
        - Story 4.7: Correction Loop Orchestration
        - Story 4.8: User Notification & Feedback
        - Story 4.9: LLM Prompt Engineering for User Communication
    - Epic 5: Conversion Agent - Format Detection
        - Story 5.1: File System Scanner
        - Story 5.2: NeuroConv Format Detection Integration
        - Story 5.3: LLM Analysis for Ambiguous Detection
    - Epic 6: Conversion Agent - Metadata & Execution
        - Story 6.1: User Metadata Collection

  - Story 6.2: Auto-Metadata Extraction
  - Story 6.3: NeuroConv Execution
  - Story 6.4: Conversion Agent Orchestration
- Epic 7: Evaluation Agent - Schema Validation & Quality Evaluation
  - Story 7.1: NWB File Information Extraction
  - Story 7.2: Schema Validation & Quality Evaluation
  - Story 7.3: Evaluation Result Processing
- Epic 8: Self-Correction Loop
  - Story 8.1: Correction Context Generation
  - Story 8.2: User Improvement Notification
  - Story 8.3: User Improvement Approval Handler
  - Story 8.3a: User Accepts File With Warnings
  - Story 8.4: Conversion Agent Self-Correction Handler
  - Story 8.5: Automatic Issue Correction
  - Story 8.6: User Input Request for Unfixable Issues
  - Story 8.7: Reconversion Orchestration
  - Story 8.8: Self-Correction Loop Termination
  - Story 8.9: User Improvement Approval UI
- Epic 9: LLM-Enhanced Evaluation Reporting
  - Story 9.1: Prompt Template for Quality Evaluation (PASSED/PASSED_WITH_ISSUES)
  - Story 9.2: Prompt Template for Correction Guidance (FAILED)
  - Story 9.3: Evaluation Agent - LLM Report Generation (PASSED/PASSED_WITH_ISSUES)
  - Story 9.4: Conversation Agent - LLM Correction Analysis (FAILED)
  - Story 9.5: PDF Report Generation with LLM Content
  - Story 9.6: JSON Context Generation with LLM Content
- Epic 10: Web API Layer
  - Story 10.1: FastAPI Application Setup
  - Story 10.2: File Upload Endpoint
  - Story 10.3: Background Task Processing
  - Story 10.4: Status API
  - Story 10.5: WebSocket Progress Streaming
  - Story 10.6: Download Endpoints
  - Story 10.7: Logs API
- Epic 11: React Web UI
  - Story 11.1: React Application Setup
  - Story 11.2: File Upload Component

    - Story 11.3: Metadata Form
    - Story 11.4: Progress View Component
    - Story 11.5: Results Display Component
    - Story 11.6: Log Viewer Component
    - Story 11.7: Basic Error Handling in UI
- Epic 12: Integration & Polish
    - Story 12.1: End-to-End Integration Test
    - Story 12.2: Sample Dataset Creation
    - Story 12.3: Installation Script
    - Story 12.4: Quick Start Script
    - Story 12.5: Error Recovery Testing
    - Story 12.6: Integration Test Timeouts
- System Requirements
    - Functional Requirements
    - Non-Functional Requirements
- Dependencies
    - External Services
    - Python Libraries
    - Frontend Libraries
    - Infrastructure
- Success Criteria
    - MVP is DONE when:
- Appendix A: API Endpoint Summary
- Appendix B: Data Schemas
    - MCP Message Schema
    - Global State Schema
    - Validation Result Schema
    - Correction Context Schema
    - API Request/Response Schemas
    - Usage Examples
        - Example 1: MCP Message Flow
        - Example 2: Validation Result Processing
        - Example 3: Global State Updates

Agentic Neurodata Conversion System - Requirements Specification

Scope: Complete end-to-end system with three-agent architecture + user-controlled error correction

---

Executive Summary

This specification defines a complete agentic system with three specialized agents:

Complete Feature Set

- ✅ **Full MCP Server Infrastructure** - JSON-RPC 2.0 protocol, agent registry, message routing
- ✅ **LLM Integration** - Anthropic Claude for intelligent decision making and report generation
- ✅ **Three Specialized Agents** - Conversation, Conversion, and Evaluation Agents with clean separation
- ✅ **Conversation Agent** - Dedicated user interaction, retry approval, input requests, LLM-powered prompts
- ✅ **Conversion Agent** - Pure technical conversion logic, format detection, NeuroConv execution
- ✅ **Evaluation Agent** - NWB validation, Inspector integration, report generation
- ✅ **User-Controlled Retry Loop** - User approves correction attempts, unlimited retries with permission
- ✅ **Web User Interface** - Modern React-based UI with real-time progress
- ✅ **Global State Management** - Single conversion tracking with stage progression
- ✅ **Intelligent Reporting** - LLM-generated scientific assessments

Scope Constraints

- 🎯 **Directory-Based Input**: Agents work on directories containing neurophysiology data (delegates format detection to NeuroConv)
- 🎯 **Single Session**: One conversion at a time (simplifies state management)
- 🎯 **Web UI Primary Interface**: React-based interface for file upload, progress tracking, and results download
- 📈 **Scalable Architecture**: Easy to add multi-session support post-MVP

System Architecture Overview

```
┌─────────────────────────────────────────────────────────┐
│                 User Interface Layer                      │
│  Web Browser (React) - File upload, Progress, Results│
└─────────────────────────────────────────────────────────┘
                  │ HTTP/WebSocket
                  ▼
┌─────────────────────────────────────────────────────────┐
│             API Gateway (FastAPI)                         │
│  REST endpoints, WebSocket, Session management           │
└─────────────────────────────────────────────────────────┘
                  │
                  ▼
┌─────────────────────────────────────────────────────────┐
│           MCP Server (Message Routing)                    │
│  Agent registry, Context management, JSON-RPC            │
└─────────────────────────────────────────────────────────┘
       │              │              │
       ▼              ▼              ▼
┌──────────────┐ ┌──────────────┐ ┌──────────────────┐
│Conversation  │ │  Conversion  │ │   Evaluation     │
│   Agent      │◄│    Agent     │◄│     Agent        │
│              │ │              │ │                  │
│ • Retry      │ │ • Format     │ │ • NWB Inspector  │
│   Approval   │ │   Detection  │ │ • Validation     │
│ • User Input │ │ • Metadata   │ │ • Result         │
│   Requests   │ │   Collection │ │   Processing     │
│ • Correction │ │ • NeuroConv  │ │ • Failure        │
│   Analysis   │ │   Execution  │ │   Context        │
│ • LLM Prompt │ │              │ │ • Report Gen     │
│   Generation │ │ • Pure       │ │                  │
│ • Loop       │ │   Technical  │ │ • NO user        │
│   Orchestr.  │ │              │ │   interaction    │
└──────────────┘ └──────────────┘ └──────────────────┘
       │              │              │
       └──────────────┴──────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────────────────┐
│             Storage & External Services                   │
│  In-Memory Store • File System • Claude API              │
└─────────────────────────────────────────────────────────┘

Three-Agent Architecture Flow:
1. User uploads → API → Conversation Agent validates metadata
2. Conversation Agent → Conversion Agent: "Convert with these params"
3. Conversion Agent detects format, converts → NWB file
4. Conversion Agent → Evaluation Agent: "Validate this NWB"
5. Evaluation Agent validates with NWB Inspector
6. IF validation PASSED (no issues at all):
   └─→ Evaluation Agent generates PDF report → User downloads NWB + PDF →
END
7. IF validation PASSED_WITH_ISSUES (has WARNING or BEST_PRACTICE issues):
   ├─→ Evaluation Agent generates improvement context
   ├─→ Evaluation Agent generates PASSED report (PDF with warnings
highlighted)
   ├─→ Evaluation Agent → Conversation Agent: "Validation passed with
warnings, here's context"
   ├─→ Conversation Agent analyzes context (categorizes issues, uses LLM)
   ├─→ Conversation Agent → User: "File is valid but has warnings. Improve?"
   └─→ User chooses:
       ├─→ IMPROVE: Continue to step 9 (enters correction loop)
       └─→ ACCEPT AS-IS: Conversation Agent finalizes, user downloads NWB +
PDF → END
8. IF validation FAILED (has CRITICAL or ERROR issues):
   ├─→ Evaluation Agent generates correction context
   ├─→ Evaluation Agent generates FAILED report (JSON)
   ├─→ Evaluation Agent → Conversation Agent: "Validation failed, here's
context"
   ├─→ Conversation Agent analyzes context (categorizes issues, uses LLM)
   ├─→ Conversation Agent → User: "Validation failed. Approve Retry?"
   └─→ User chooses:
       ├─→ APPROVE: Continue to step 9 (enters correction loop)
       └─→ DECLINE: Conversation Agent finalizes, user downloads NWB + JSON
report → END
9. IF user approves improvement/retry:
   ├─→ Conversation Agent identifies auto-fixable issues
   ├─→ Conversation Agent identifies issues needing user input
   ├─→ IF needs user input:
   │   ├─→ Conversation Agent generates prompts (using LLM)
   │   ├─→ Conversation Agent → User: "Please provide X (example: ...)"
   │   └─→ User provides data
   ├─→ Conversation Agent → Conversion Agent: "Reconvert with these fixes +
user data"
   ├─→ Conversion Agent applies corrections and reconverts
   └─→ Loop back to step 4 (unlimited retries with user permission)

Key Architectural Benefits:
• Conversation Agent owns user interaction logic (retry approval, input
requests)
• Conversion Agent is pure technical conversion (no user interaction)
• Evaluation Agent is pure validation (no user interaction)
• Clean separation allows independent scaling, testing, and reuse
```

Personas & Stakeholders

This specification uses **three consistent personas** to clarify who benefits from each feature.

Persona 1: User (Data Scientist / Researcher)

- **Role**: "As a user"
- **Goals**: Convert neurophysiology data to NWB format, validate data quality, receive scientifically meaningful reports
- **Technical Level**: Intermediate (understands data formats, not necessarily software architecture)
- **Examples**: Upload files (Epic 4), view progress (Epic 10), download results (Epic 11), approve retries (Epic 8)

Persona 2: System (Technical Requirements)

- **Role**: "As the system"
- **Definition**: Technical/architectural requirements that enable user-facing features
- **Not a real user**: Represents system components, agent design, infrastructure, and internal protocols
- **Examples**: MCP server (Epic 1), format detection (Epic 5), conversion logic (Epic 6), evaluation (Epic 7), LLM reporting (Epic 9)
- **Note**: Agent responsibilities are system requirements, not user needs. Stories say "As the system" instead of "As the Evaluation Agent"

Persona 3: Developer/Maintainer

- **Role**: "As a developer"
- **Goals**: Implement, test, debug, and maintain the system
- **Technical Level**: Advanced (full-stack developer with AI/ML knowledge)
- **Examples**: Create sample datasets (Epic 12), write integration tests (Epic 12), set up infrastructure (Epic 12)

User Stories by Epic

Epic 1: MCP Server Infrastructure

Story 1.1: MCP Server Foundation

Depends on: None (foundational)

As the system **I want** a Model Context Protocol server that can register and manage agents **So that** agents can communicate through a standardized protocol

Acceptance Criteria:

- ☐ Server accepts agent registrations with name, handler, and capabilities
- ☐ Server maintains active registry of all registered agents
- ☐ Server provides agent discovery (list all agents)
- ☐ Server can unregister agents
- ☐ Server logs all registration/unregistration events
- ☐ Agent registry accessible via API call

Priority: Critical

Story 1.2: Message Routing System

Depends on: Story 1.1

As the system **I want** to route messages between agents based on target specification **So that** agents can invoke each other's capabilities

Acceptance Criteria:

- ☐ Messages contain target_agent, action, and context fields
- ☐ Server validates target agent exists before routing
- ☐ Server invokes target agent's handler with message
- ☐ Server returns agent response to caller
- ☐ Server handles routing failures gracefully
- ☐ All message routing logged with timestamps

Priority: Critical

Story 1.3: Context Management

Depends on: Story 1.2, Story 2.1

As the system **I want** to attach global state context to every message **So that** agents have complete information for decision making

Acceptance Criteria:

- ☐ Server retrieves global state data
- ☐ Server attaches state context to message
- ☐ Context includes status, metadata, stages, and logs
- ☐ Agents can update context via server
- ☐ Context changes reflected in global state immediately
- ☐ Context accessible by all agents

Priority: High

Epic 2: Global State Management (Single Session)

Story 2.1: Global State Object

Depends on: None (foundational)

As a developer **I want** a single global state object to track the current conversion **So that** state management is simple and efficient

Acceptance Criteria:

- ☐ Global state variable stores: status, validation_status, input_path, output_path, metadata, logs, stages, timestamps
- ☐ Status tracked: idle, processing, completed, failed
- ☐ Validation_status tracked: null (not yet validated), passed, passed_accepted, passed_improved, failed_user_declined, failed_user_abandoned
- ☐ State initialized to idle on startup
- ☐ State resets after each conversion completes (all fields including validation_status)
- ☐ State is JSON-serializable for debugging
- ☐ No thread-safety needed (single conversion at a time)

Priority: Critical

Story 2.2: Stage Tracking

Depends on: Story 2.1

As the system **I want** to track conversion pipeline stages **So that** users see where they are in the workflow

Acceptance Criteria:

- [ ] Global state tracks stages: conversion, evaluation, report_generation
- [ ] Each stage has: name, status (pending/in_progress/completed/failed), start_time, end_time
- [ ] Stage results stored (output_path, error_message, metadata)
- [ ] Stage updates reflected in global status
- [ ] Current stage queryable via API
- [ ] Stage information sent via WebSocket for UI updates

Priority: High

Epic 3: LLM Service Foundation

Story 3.1: LLM Service Abstract Interface

Depends on: None (foundational)

As a developer **I want** an abstract LLM service interface **So that** I can swap between different LLM providers (Claude, GPT-4)

Acceptance Criteria:

- [ ] Abstract base class defines complete() and chat() methods
- [ ] Token counting and truncation utilities
- [ ] Error handling standardized across providers
- [ ] Configuration via environment variables
- [ ] Logging of all LLM calls
- [ ] Token usage tracking

Priority: High

Story 3.2: Anthropic Claude Integration

Depends on: Story 3.1

As the system **I want** to integrate with Anthropic Claude API **So that** I can leverage LLM for intelligent analysis

Acceptance Criteria:

- [ ] Service authenticates with API key
- [ ] Service sends prompts and receives completions
- [ ] Service handles API errors (rate limits, timeouts, network errors)
- [ ] Service retries transient failures
- [ ] Service logs token usage for cost tracking
- [ ] Service supports both completion and chat modes

Priority: Critical

Epic 4: Conversation Agent - User Interaction

Story 4.1: Conversation Agent Foundation

Depends on: Story 1.2, Story 2.1, Story 3.2

As the system **I want** a dedicated Conversation Agent for user interaction **So that** user communication is separated from technical conversion logic

Acceptance Criteria:

- [ ] Agent registers with MCP server as "conversation_agent"
- [ ] Agent exposes MCP tools for user interaction
- [ ] Agent maintains user session state (waiting_for_approval, waiting_for_input, processing)
- [ ] Agent has access to LLM service for prompt generation
- [ ] Agent can send messages to Conversion and Evaluation agents via MCP

- ☐ Agent logs all user interactions
- ☐ Agent rejects concurrent requests with clear error (single session constraint)

Priority: Critical

Story 4.2: Initial Metadata Validation Handler

Depends on: Story 4.1

As the system **I want** to validate user-provided metadata before conversion **So that** I catch missing required fields early

Acceptance Criteria:

- ☐ Agent receives upload request with files + metadata from API
- ☐ Agent validates required fields per NWB schema: subject_id, species, session_description, session_start_time
- ☐ Agent validates recommended fields: experimenter, institution, lab
- ☐ Agent validates optional fields if provided (age, sex, weight)
- ☐ Agent checks metadata format and types (ISO 8601 for dates, alphanumeric for IDs)
- ☐ IF validation fails: Agent generates user-friendly error message
- ☐ IF validation passes: Agent forwards to Conversion Agent
- ☐ Validation completes in <1 second

Priority: Critical

Story 4.3: Improvement Approval Handler (Deprecated - See Stories 8.2, 8.3, 8.3a)

Depends on: Story 4.1, Story 7.3

Note: This story's functionality has been split into:

- **Story 8.2**: User Improvement Notification (Conversation Agent notifies user)
- **Story 8.3**: User Improvement Approval Handler (System handles decision)
- **Story 8.3a**: User Accepts File With Warnings (User story for "Accept As-Is" path)

This entry preserved for reference. Implementation should follow Stories 8.2, 8.3, and 8.3a.

Original Acceptance Criteria (now superseded):

- [ ] Agent receives context from Evaluation Agent via MCP (FAILED or PASSED_WITH_ISSUES)
- [ ] Agent analyzes correction context (categorizes issues by severity)
- [ ] For FAILED status:
  - Agent generates failure summary with CRITICAL/ERROR issues
  - "Auto-fixable issues" list with descriptions
  - "Requires your input" list with descriptions
  - Agent sends message: "Validation failed. Review issues and approve retry?"
- [ ] For PASSED_WITH_ISSUES status:
  - Agent generates improvement summary with WARNING/BEST_PRACTICE issues
  - "Auto-fixable improvements" list with descriptions
  - "Requires your input for best results" list with descriptions
  - Agent sends message: "File is valid but has warnings. Would you like to improve?"
- [ ] Agent sends approval request to API/UI
- [ ] Agent waits indefinitely for user decision (no timeout)
- [ ] Agent logs user decision (approve/decline/accept-as-is) with timestamp and status type
- [ ] IF user approves improvement/retry: Forward corrections to Conversion Agent
- [ ] IF user declines/accepts-as-is: Finalize session with appropriate validation_status

Priority: Critical (Superseded)

Story 4.4: Correction Context Analysis with LLM

Depends on: Story 3.2

As the system **I want** to use LLM to analyze validation failures **So that** I can explain errors clearly to users

Acceptance Criteria:

- [ ] Agent sends validation issues to LLM with context
- [ ] LLM prompt requests:
    - Plain language explanation of each issue
    - Whether issue is auto-fixable or needs user input
    - Suggested fix strategies
    - Impact on final NWB file
- [ ] Agent parses LLM response into structured format
- [ ] Agent throws clear error if LLM unavailable with format:
    - Root cause: "Claude API unavailable: [specific error code and message]"
    - Retry strategy: "Retry in 5 minutes (rate limit) or check ANTHROPIC_API_KEY environment variable"
    - Support: "Error ID: [uuid] - Include this in support requests"
- [ ] Analysis completes in <5 seconds
- [ ] LLM token usage tracked and logged

Note: LLM failures during correction analysis are **critical errors** that stop the correction loop. For optional LLM usage (like Story 5.3 format detection), the system degrades gracefully.

Priority: High

Story 4.5: User Input Request Generator with LLM

Depends on: Story 3.2

As the system **I want** to generate clear prompts for missing/incorrect data **So that** users understand what to provide and why

Acceptance Criteria:

- [ ] Agent identifies fields requiring user input from correction context
- [ ] Agent uses LLM to generate contextual prompts:
    - Clear question: "What is X?"
    - Why it's needed: "This is required for Y"
    - Example values: "e.g., 'mouse_001', 'rat_042'"
    - Validation rules: "Must be alphanumeric, max 50 chars"
- [ ] Agent groups related prompts (e.g., all subject fields together)

- [ ] Agent sends prompts to API/UI via MCP
- [ ] Agent waits for user response
- [ ] Prompt generation completes in <3 seconds per field

Priority: High

Story 4.6: User Input Validation

Depends on: Story 4.5

As the system **I want** to validate user-provided input before using it **So that** corrections don't introduce new errors

Acceptance Criteria:

- [ ] Agent receives user input from API
- [ ] Agent validates input against field requirements:
    - Type checking (string, number, date, etc.)
    - Format validation (e.g., ISO dates, alphanumeric IDs)
    - Length/range validation
    - Enum validation (e.g., species from approved list)
- [ ] Agent uses LLM to validate domain-specific constraints
- [ ] IF validation fails: Agent generates error message and re-prompts
- [ ] IF validation passes: Agent stores validated input
- [ ] Agent allows user to cancel/skip optional fields
- [ ] Validation logged for audit trail

Priority: High

Story 4.7: Correction Loop Orchestration

Depends on: Story 4.1, Story 4.3

As the system **I want** to orchestrate the correction loop across agents **So that** retry attempts are coordinated properly

Acceptance Criteria:

- [ ] Agent maintains correction loop state:
    - Current attempt number
    - Issues identified
    - Issues fixed automatically
    - Issues fixed with user input
    - Pending user input requests
- [ ] Agent coordinates message flow:
    - Evaluation Agent → Conversation Agent (failure context)
    - Conversation Agent → User (approval/input requests)
    - User → Conversation Agent (decisions/data)
    - Conversation Agent → Conversion Agent (correction params)
    - Conversion Agent → Evaluation Agent (reconverted file)
- [ ] Agent tracks correction history for each session
- [ ] Agent prevents duplicate correction attempts by detecting "no progress":
    - Same exact validation errors between attempts (error codes + locations match)
    - No user input provided since last attempt
    - No auto-corrections applied since last attempt
    - Agent warns user: "No changes detected since last attempt. Retry will likely produce same errors."
- [ ] Agent respects user termination requests
- [ ] Unlimited retry attempts with user permission

Priority: Critical

Story 4.8: User Notification & Feedback

Depends on: Story 4.1

As the system **I want** to keep users informed throughout the process **So that** users understand system status at all times

Acceptance Criteria:

- [ ] Agent sends real-time notifications via WebSocket:
    - "Conversion started"
    - "Validation in progress"
    - "Validation failed - review needed" (for FAILED status)

    - "Validation passed with warnings - review recommended" (for PASSED_WITH_ISSUES status)
    - "Validation passed - no issues found!" (for PASSED status with no issues)
    - "Applying automatic corrections"
    - "Applying improvements to resolve warnings"
    - "Awaiting your input for X"
    - "Reconverting with corrections"
    - "Re-validating improved file"
    - "All warnings resolved!" (for PASSED after improvement from PASSED_WITH_ISSUES)
    - "All errors fixed!" (for PASSED after correction from FAILED)
- [ ] Agent includes progress indicators (percentages, stages)
- [ ] Agent provides actionable next steps in each notification
- [ ] Agent handles notification failures gracefully
- [ ] Notifications include timestamps and attempt numbers
- [ ] Notifications indicate validation status type (PASSED/PASSED_WITH_ISSUES/FAILED)

Priority: High

Story 4.9: LLM Prompt Engineering for User Communication

Depends on: Story 3.2

As a developer **I want** well-structured LLM prompts for user-facing messages **So that** communication is clear, helpful, and domain-appropriate

Acceptance Criteria:

- [ ] Prompts structured with:
    - System role: "You are a helpful neuroscience data assistant"
    - Context: Validation issue details
    - Task: Explain issue or generate input prompt
    - Constraints: Plain language, no jargon unless necessary, max 150 words
    - Output format: JSON with structured fields
- [ ] Prompts include examples of good explanations

- [ ] Prompts request uncertainty indication ("I'm not sure, but...")
- [ ] Prompts fit within Claude's context window
- [ ] Prompts logged for refinement

Priority: Medium

Epic 5: Conversion Agent - Format Detection

Story 5.1: File System Scanner

Depends on: None (foundational)

As the system **I want** to scan user-provided paths and catalog all files **So that** I can analyze data structure for format detection

Acceptance Criteria:

- [ ] Agent accepts file path or directory path
- [ ] Agent recursively scans directories
- [ ] Agent catalogs files by extension
- [ ] Agent computes file sizes
- [ ] Agent generates file listing for LLM
- [ ] Agent handles permission errors gracefully

Priority: High

Story 5.2: NeuroConv Format Detection Integration

Depends on: Story 5.1

As the system **I want** to use NeuroConv's built-in format detection capabilities **So that** agents can handle any format NeuroConv supports without manual pattern matching

Acceptance Criteria:

- ☐ Agent uses NeuroConv's automatic data interface detection
- ☐ Agent passes directory path to NeuroConv for analysis
- ☐ Agent receives detected interfaces and their confidence scores from NeuroConv
- ☐ Agent logs detected format(s) with confidence levels
- ☐ Agent handles NeuroConv detection failures with clear error messages
- ☐ Detection leverages NeuroConv's native capabilities (no manual file pattern matching)
- ☐ Detection completes in <5 seconds per directory
- ☐ Agent can query NeuroConv documentation via MCP server when detection is ambiguous

Priority: Critical

Story 5.3: LLM Analysis for Ambiguous Detection

Depends on: Story 5.2, Story 3.2

As the system **I want** LLM to analyze ambiguous detection results from NeuroConv **So that** agents can make informed decisions when multiple formats match

Acceptance Criteria:

- ☐ Agent invokes LLM only when NeuroConv returns multiple possible formats
- ☐ LLM prompt includes directory structure, file listing, and NeuroConv's candidate interfaces
- ☐ LLM prompt can reference NeuroConv documentation via MCP server
- ☐ LLM response selects most likely interface with reasoning
- ☐ Agent logs LLM reasoning for transparency
- ☐ **Graceful degradation**: Agent proceeds with NeuroConv's highest-confidence result if LLM unavailable (no exception raised)
- ☐ Agent logs warning when LLM unavailable: "Format detection using NeuroConv default (LLM unavailable)"
- ☐ Analysis completes in <10 seconds

Note: This is **optional LLM usage** for enhancement. LLM failure does NOT stop the conversion (unlike Story 4.4 where LLM is critical).

Priority: Medium

Epic 6: Conversion Agent - Metadata & Execution

Story 6.1: User Metadata Collection

Depends on: Story 4.1

As the system **I want** to collect required NWB metadata from user **So that** converted files have complete information

Acceptance Criteria:

- [ ] Agent receives required fields from Conversation Agent: subject_id, species, session_description, session_start_time
- [ ] Agent validates subject_id (non-empty, alphanumeric)
- [ ] Agent validates species (non-empty string from approved taxonomy)
- [ ] Agent validates session_description (non-empty string)
- [ ] Agent validates session_start_time (ISO 8601 format)
- [ ] Agent returns clear validation errors to Conversation Agent
- [ ] Validated metadata stored in session

Priority: Critical

Story 6.2: Auto-Metadata Extraction

Depends on: Story 5.2

As the system **I want** to extract technical metadata from data files **So that** users don't need to provide it manually

Acceptance Criteria:

- [ ] Agent extracts sampling rate from file headers
- [ ] Agent extracts channel count

- ☐ Agent extracts recording duration
- ☐ Agent extracts data type and bit depth
- ☐ Agent handles missing metadata fields gracefully
- ☐ Extracted metadata logged and stored

Priority: High

Story 6.3: NeuroConv Execution

Depends on: Story 6.1, Story 6.2, Story 5.2

As the system **I want** agents to execute NeuroConv conversion using auto-detected interfaces **So that** any NeuroConv-supported format converts to NWB standardized format

Acceptance Criteria:

- ☐ Agent uses interface(s) detected by NeuroConv in Story 5.2
- ☐ Agent initializes NeuroConv converter with detected interface and directory path
- ☐ Agent merges auto-extracted metadata with user-provided metadata
- ☐ Agent runs conversion to specified output path (.nwb file)
- ☐ Agent raises clear exceptions for conversion failures (defensive errors, no silent failures)
- ☐ Agent verifies output file created and is readable by PyNWB
- ☐ Agent computes SHA256 checksum of output file
- ☐ Conversion progress logged with: format detected, interface used, file size, duration
- ☐ Agent logs full NeuroConv error messages without modification (aids debugging)
- ☐ **Error Recovery**: On conversion failure, agent sends error details to Conversation Agent via MCP
- ☐ Conversation Agent receives error and notifies user with diagnostics
- ☐ Global state marked as FAILED with error details stored

Priority: Critical

Story 6.4: Conversion Agent Orchestration

Depends on: Story 6.3, Story 1.2

As the system **I want** to orchestrate the complete conversion workflow **So that** conversion happens autonomously from start to finish

Acceptance Criteria:

- [ ] Agent receives data path, metadata, output directory from MCP
- [ ] Agent executes: scan → detect → validate metadata → extract → convert → verify
- [ ] Agent updates session stage status at each step
- [ ] Agent logs all major actions
- [ ] Agent returns output path and metadata on success
- [ ] Agent returns detailed error on failure

Priority: Critical

Epic 7: Evaluation Agent - Schema Validation & Quality Evaluation

Story 7.1: NWB File Information Extraction

Depends on: Story 6.3

As the system **I want** to extract comprehensive information from NWB files **So that** reports contain complete file characterization

Acceptance Criteria:

- [ ] Agent extracts top-level attributes (NWB version, creation date, identifier)
- [ ] Agent extracts all /general metadata (session info, experimenter, institution)
- [ ] Agent extracts subject information (ID, species, age, sex)
- [ ] Agent extracts device information
- [ ] Agent extracts electrode information (groups, tables)
- [ ] Agent extracts acquisition data inventory (names, types, shapes, sizes)
- [ ] Agent extracts processing modules
- [ ] Agent computes file statistics (size, temporal coverage)

- ☐ Agent handles missing optional fields gracefully
- ☐ Extraction completes in <30 seconds for typical files

Priority: Critical

Story 7.2: Schema Validation & Quality Evaluation

Depends on: Story 7.1

As the system **I want** to perform both schema validation and quality evaluation **So that** files are both NWB-compliant and scientifically useful

Acceptance Criteria:

- ☐ **Schema Validation**: Agent verifies file is readable by PyNWB (confirms NWB schema compliance)
- ☐ **Quality Evaluation**: Agent runs NWB Inspector with all checks for quality assessment
- ☐ Agent captures all Inspector issues (these are quality warnings, not schema violations)
- ☐ Agent categorizes issues by severity (CRITICAL, ERROR, WARNING, BEST_PRACTICE)
- ☐ Agent extracts check name, message, location for each issue
- ☐ Agent determines overall evaluation status:
  - FAILED: If any CRITICAL or ERROR issues present (poor quality, may not be usable)
  - PASSED_WITH_ISSUES: If no CRITICAL/ERROR but has WARNING or BEST_PRACTICE issues (usable but improvable)
  - PASSED: If no issues at all (high quality)
- ☐ Agent raises exceptions for Inspector timeouts or errors (defensive approach)
- ☐ Evaluation completes in <2 minutes for typical files

Note: `overall_status` (PASSED/PASSED_WITH_ISSUES/FAILED) is the **evaluation result** from NWB Inspector. The global state's `validation_status` (from Story 2.1) tracks the **final session outcome** including user decisions (e.g., "passed_accepted", "passed_improved", "failed_user_declined").

Priority: Critical

Story 7.3: Evaluation Result Processing

Depends on: Story 7.2, Story 2.1

As the system **I want** to process evaluation results into structured format **So that** downstream agents can analyze and report on quality

Acceptance Criteria:

- ☐ Agent counts issues by severity (CRITICAL, ERROR, WARNING, BEST_PRACTICE)
- ☐ Agent groups issues by category (missing metadata, incorrect units, etc.)
- ☐ Agent identifies critical issues for FAILED status
- ☐ Agent generates summary statistics (total issues, file size, completeness score)
- ☐ Agent stores evaluation results in global state
- ☐ Results are JSON-serializable and accessible via API
- ☐ Agent preserves all Inspector output for debugging (full logs)

Priority: High

Epic 8: Self-Correction Loop

Story 8.1: Correction Context Generation

Depends on: Story 7.3, Story 3.2

As the system **I want** to generate actionable correction context when validation has issues **So that** downstream agents can present options to users and orchestrate fixes

Acceptance Criteria:

- ☐ Evaluation Agent generates correction context when status is FAILED or PASSED_WITH_ISSUES
- ☐ For FAILED status:

    - Context includes all CRITICAL and ERROR issues with details
    - Agent generates FAILED report (JSON) with human-readable issue descriptions
- ☐ For PASSED_WITH_ISSUES status:
    - Context includes all WARNING and BEST_PRACTICE issues with details
    - Agent generates PASSED report (PDF) with issue highlights
- ☐ Context categorizes issues by type (missing data, incorrect metadata, schema violations, etc.)
- ☐ Context identifies auto-fixable issues vs. user-input-required issues
- ☐ Context includes specific file locations and field names for each issue
- ☐ Context is JSON-serializable and well-structured
- ☐ Evaluation Agent sends context to Conversation Agent via MCP (does NOT interact with user directly)

Priority: Critical

Story 8.2: User Improvement Notification

Depends on: Story 8.1, Story 4.1

As the system **I want** to notify users about validation results and improvement options
So that users can make informed decisions about correction attempts

Acceptance Criteria:

- ☐ Conversation Agent receives correction context from Evaluation Agent via MCP
- ☐ Agent analyzes context (categorizes issues by severity)
- ☐ For FAILED status:
    - Agent generates failure summary with CRITICAL/ERROR issues
    - "Auto-fixable issues" list with descriptions
    - "Requires your input" list with descriptions
    - Agent sends message: "Validation failed. Review issues and approve retry?"
- ☐ For PASSED_WITH_ISSUES status:
    - Agent generates improvement summary with WARNING/BEST_PRACTICE issues
    - "Auto-fixable improvements" list with descriptions
    - "Requires your input for best results" list with descriptions
    - Agent sends message: "File is valid but has warnings. Would you like to improve?"

- [ ] Agent sends notification to API/UI via WebSocket
- [ ] Agent waits indefinitely for user decision (no timeout)
- [ ] Agent logs notification sent with timestamp and status type

Priority: Critical

Story 8.3: User Improvement Approval Handler

Depends on: Story 8.2

As the system **I want** to handle user decision on improvement approval **So that** correction only proceeds with user consent

Acceptance Criteria:

- [ ] System displays validation summary to user (FAILED or PASSED_WITH_ISSUES)
- [ ] System shows categorized list of issues (auto-fixable vs. needs input)
- [ ] For FAILED status:
    - System presents "Approve Retry" and "Decline Retry" options
    - IF user approves: Send correction context to Conversion Agent via MCP
    - IF user declines: Finalize session with FAILED status, provide NWB + JSON report for download
- [ ] For PASSED_WITH_ISSUES status:
    - System presents "Improve File" and "Accept As-Is" options
    - IF user approves improvement: Send correction context to Conversion Agent via MCP (same as FAILED flow)
    - IF user declines: Finalize session with PASSED status, provide NWB + PDF report for download (file is already acceptable)
- [ ] User can review full report before deciding
- [ ] No timeout on user decision (wait indefinitely)
- [ ] Decision logged in session history

Priority: Critical

Story 8.3a: User Accepts File With Warnings

Depends on: Story 8.2

As a user **I want** to accept my file as valid despite warnings **So that** I can proceed with a usable file without further improvement

Acceptance Criteria:

- [ ] User sees "Accept As-Is" option when validation status is PASSED_WITH_ISSUES
- [ ] User can review PDF report with all warnings before deciding
- [ ] User can download NWB + PDF report immediately after accepting
- [ ] System sets global validation_status to "passed_accepted"
- [ ] No correction loop initiated (session ends successfully)
- [ ] Decision logged: "User accepted file with N warnings at [timestamp]"
- [ ] UI displays confirmation: "File accepted. Download ready."

Priority: High

Story 8.4: Conversion Agent Self-Correction Handler

Depends on: Story 8.3, Story 1.2

As the system **I want** to receive and process failure context when user approves retry **So that** I can automatically correct conversion issues

Acceptance Criteria:

- [ ] Agent receives correction context via MCP ONLY after user approval
- [ ] Agent analyzes correction context to determine fix strategy
- [ ] Agent distinguishes between auto-fixable and user-input-required issues
- [ ] Agent stores correction context in global state
- [ ] Agent updates stage to "correction_in_progress"
- [ ] Agent logs all correction attempts with attempt number

Priority: Critical

Story 8.5: Automatic Issue Correction

Depends on: Story 8.4

As the system **I want** to automatically fix issues that don't require user input **So that** the system is truly autonomous

Acceptance Criteria:

- ☐ Agent identifies auto-fixable issue types:
    - Missing optional metadata (use defaults or infer from data)
    - Incorrect data types (auto-convert when safe)
    - Missing timestamps (infer from file metadata)
    - Incorrect units (convert using standard mappings)
    - Missing descriptions (generate from field names)
- ☐ Agent applies fixes to conversion parameters
- ☐ Agent logs all automatic corrections made
- ☐ Agent reconverts with corrected parameters
- ☐ Agent limits automatic fixes to safe operations only

Priority: High

Story 8.6: User Input Request for Unfixable Issues

Depends on: Story 8.4, Story 4.5

As the system **I want** to request user input for issues I cannot fix automatically **So that** conversions can succeed even with complex problems

Acceptance Criteria:

- ☐ Agent identifies issues requiring user input:
    - Missing required metadata (subject_id, species, etc.)
    - Ambiguous data interpretations
    - Multiple possible fix strategies
    - Domain-specific knowledge required
- ☐ Agent generates clear, specific prompts for user
- ☐ Agent includes context and examples in prompts
- ☐ Agent sends user input request via MCP to API layer

- ☐ Agent waits for user response before proceeding
- ☐ Agent validates user-provided data
- ☐ Agent incorporates user input into reconversion

Priority: High

Story 8.7: Reconversion Orchestration

Depends on: Story 8.5, Story 8.6, Story 6.3

As the system **I want** to orchestrate the reconversion process after applying fixes **So that** the self-correction loop completes successfully

Acceptance Criteria:

- ☐ Agent applies all automatic fixes to conversion parameters
- ☐ Agent incorporates any user-provided data
- ☐ Agent invokes NeuroConv with corrected parameters
- ☐ Agent generates new NWB file (versioned: original.nwb, original_v2.nwb, etc.)
- ☐ Agent computes SHA256 checksum for each generated NWB file
- ☐ Agent preserves original file as immutable version (no overwrites)
- ☐ Agent creates new version only if reconversion succeeds
- ☐ If reconversion fails: Original file remains downloadable with checksum verification
- ☐ Agent stores checksums in global state for integrity verification
- ☐ Agent sends new NWB file to Evaluation Agent for revalidation
- ☐ Agent tracks attempt number (no maximum limit - continues until user declines or PASSED)
- ☐ Agent updates global state with reconversion progress
- ☐ Each reconversion triggers new user approval cycle if validation fails again

Priority: Critical

Story 8.8: Self-Correction Loop Termination

Depends on: Story 8.7

As the system **I want** to properly terminate the self-correction loop **So that** loops end appropriately based on user decisions

Acceptance Criteria:

- [ ] Loop terminates on PASSED validation status (no issues - success case)
- [ ] Loop terminates on PASSED_WITH_ISSUES when user chooses "Accept As-Is" (acceptable file)
- [ ] Loop terminates when user declines retry approval on FAILED (user choice)
- [ ] Loop terminates if user cancels data input request (user abandons)
- [ ] No automatic termination based on attempt count (unlimited retries with user permission)
- [ ] Final status reflects outcome:
  - "passed" - validation succeeded with no issues
  - "passed_accepted" - validation passed with warnings, user accepted as-is
  - "passed_improved" - validation passed after improvement loop
  - "failed_user_declined" - user chose not to retry failed validation
  - "failed_user_abandoned" - user cancelled during data input
- [ ] All loop iterations logged for debugging with timestamps
- [ ] Final report includes complete correction history with all attempts

Priority: High

Story 8.9: User Improvement Approval UI

Depends on: Story 8.2

As a user **I want** to see validation results and decide whether to improve the file **So that** I control the improvement process

Acceptance Criteria:

- [ ] When validation FAILED, UI shows "Validation Failed" with clear summary:
  - Prominent red banner with CRITICAL/ERROR count
  - "Approve Retry" and "Decline Retry" buttons
  - "Decline Retry" allows downloading NWB file + JSON report
- [ ] When validation PASSED_WITH_ISSUES, UI shows "Validation Passed with Warnings":

    - Prominent yellow banner with WARNING/BEST_PRACTICE count
    - "Improve File" and "Accept As-Is" buttons
    - "Accept As-Is" allows downloading NWB file + PDF report immediately
    - Message: "File is valid and usable, but can be improved"
- [ ] UI displays categorized issues for both cases:
    - "Auto-fixable issues" (system will handle automatically)
    - "Requires your input" (you'll be prompted for data)
- [ ] Action buttons include attempt count if multiple attempts made (e.g., "Retry Again (Attempt 3)" or "Improve Again (Attempt 2)")
- [ ] UI provides expandable view of full validation report
- [ ] During correction: UI shows "Improvement in Progress (Attempt N)" or "Correction in Progress (Attempt N)"
- [ ] UI displays which issues are being corrected in real-time
- [ ] When user input needed: Modal with clear prompts and examples
- [ ] Results view shows complete correction history for all attempts
- [ ] Final success shows different messages:
    - PASSED (no issues): "Perfect! No issues found."
    - PASSED (after improvement): "Success! All warnings resolved."

Priority: High

Epic 9: LLM-Enhanced Evaluation Reporting

Pattern: This epic defines how LLM analysis integrates into evaluation reporting:

1. **Stories 9.1-9.2**: Define prompt templates (system configuration)
2. **Stories 9.3-9.4**: Implement agent logic that uses templates
3. **Stories 9.5-9.6**: Format output (PDF for PASSED/PASSED_WITH_ISSUES, JSON for FAILED)

Story 9.1: Prompt Template for Quality Evaluation (PASSED/PASSED_WITH_ISSUES)

Depends on: Story 3.2

As the system **I want** a reusable prompt template for LLM analysis of quality evaluation results **So that** the Evaluation Agent can generate consistent scientific assessments

Acceptance Criteria:

- ☐ Template stored as configuration (e.g., YAML, JSON, or Python f-string)
- ☐ Template includes placeholders for: file_info, evaluation_status, issues_list, issue_counts
- ☐ Template structure:

  ```
  System Role: "You are a neuroscience data quality analyst"
  Context: {file_info}, {evaluation_status}, {issues_breakdown}
  Task: Analyze quality evaluation results
  Output Format: JSON with fields {executive_summary, quality_assessment,
  recommendations}
  Guidelines: Ground in data, use neuroscience terminology, be specific
  ```

- ☐ Template has variants for PASSED vs PASSED_WITH_ISSUES:
  - **PASSED**: Emphasize completeness and quality
  - **PASSED_WITH_ISSUES**: Analyze each warning (scientific meaning, impact, improvement value)
- ☐ Template fits within Claude's context window (with truncation strategy if needed)
- ☐ Template versioned in codebase (not hardcoded in agent logic)

Priority: High

Story 9.2: Prompt Template for Correction Guidance (FAILED)

Depends on: Story 3.2

As the system **I want** a reusable prompt template for LLM analysis of failed evaluation results **So that** the Conversation Agent can generate actionable correction guidance

Acceptance Criteria:

- ☐ Template stored as configuration (e.g., `src/prompts/evaluation_failed.yaml`)
- ☐ Template includes placeholders for: critical_issues, error_details, file_context
- ☐ Template structure:

```
System Role: "You are a helpful NWB data quality assistant"
Context: {file_info}, {critical_issues}, {error_breakdown}
Task: Generate actionable fix guidance
Output Format: JSON with fields {issue_analysis, fix_roadmap,
auto_fixable, user_input_needed}
Guidelines: Plain language, step-by-step instructions, encouraging tone
```

- ☐ Template requests per-issue analysis:
  - What the issue is (plain language)
  - Why it matters
  - How to fix it
  - Whether auto-fixable or needs user input
- ☐ Template requests prioritized fix roadmap with dependencies
- ☐ Template optimized for Conversation Agent to parse and act on
- ☐ Template versioned in codebase

Priority: High

Story 9.3: Evaluation Agent - LLM Report Generation (PASSED/PASSED_WITH_ISSUES)

Depends on: Story 9.1, Story 7.3

As the system **I want** to invoke LLM with the quality evaluation prompt template **So that** I can generate scientific assessment reports

Acceptance Criteria:

- ☐ Agent loads prompt template from Story 9.1 (`src/prompts/evaluation_passed.yaml`)
- ☐ Agent populates template with evaluation results from Story 7.3
- ☐ Agent calls LLM service (Story 3.2) with populated prompt
- ☐ Agent receives and validates JSON response structure

- ☐ Agent parses response into `EvaluationReport` schema (defined in Appendix C)
- ☐ Agent raises exception if LLM fails (defensive error handling):
    - Include full API error details
    - Include prompt that was sent (for debugging)
    - Log to structured logs
- ☐ Agent logs LLM token usage for cost tracking
- ☐ Operation completes or raises exception (no timeout enforcement - let LLM take time needed)

Priority: High

Story 9.4: Conversation Agent - LLM Correction Analysis (FAILED)

Depends on: Story 9.2, Story 7.3, Story 4.4

As the system **I want** to invoke LLM with the correction guidance prompt template **So that** I can orchestrate the self-correction loop with actionable fix strategies

Acceptance Criteria:

- ☐ Agent loads prompt template from Story 9.2 (`src/prompts/evaluation_failed.yaml`)
- ☐ Agent populates template with failed evaluation results from Story 7.3
- ☐ Agent calls LLM service (Story 3.2) with populated prompt
- ☐ Agent receives and validates JSON response structure
- ☐ Agent parses response into `CorrectionContext` schema (Appendix C)
- ☐ Agent extracts:
    - `auto_fixable_issues`: List of issues Conversion Agent can fix automatically
    - `user_input_required_issues`: List of issues needing user data
    - `fix_roadmap`: Prioritized steps with dependencies
- ☐ Agent raises exception if LLM fails (defensive error handling)
- ☐ Agent logs LLM token usage
- ☐ Agent uses parsed output to drive correction loop (Story 8.3)

Priority: High

Story 9.5: PDF Report Generation with LLM Content

Depends on: Story 9.3

As the system **I want** to generate professional PDF reports with LLM analysis for PASSED/PASSED_WITH_ISSUES files **So that** users get comprehensive, human-readable assessments

Acceptance Criteria:

- [ ] Agent uses LLM-generated content from Story 9.3 (`EvaluationReport` schema)
- [ ] PDF includes:
  - Cover page: status, file info, date, NWB version
  - Executive summary from LLM analysis
  - File information table (metadata, data contents, statistics)
  - Evaluation results table (issue counts by severity)
  - Issues list (if PASSED_WITH_ISSUES): each warning with LLM explanation
  - LLM analysis sections: quality assessment, scientific insights, recommendations
  - Conclusions
- [ ] PDF professionally formatted with sections, tables, page numbers
- [ ] PDF filename: `<nwb_filename>_evaluation_report.pdf`
- [ ] PDF written to output directory alongside NWB file
- [ ] PDF path stored in global state for download/access
- [ ] Agent raises exception if PDF generation fails (defensive)

Priority: Critical

Story 9.6: JSON Context Generation with LLM Content

Depends on: Story 9.4

As the system **I want** to generate structured JSON with LLM guidance for FAILED files **So that** users have machine-readable fix instructions

Acceptance Criteria:

- ☐ Agent uses LLM-generated content from Story 9.4 (`CorrectionContext` schema)
- ☐ JSON includes:
    - Evaluation metadata (ID, status, timestamp)
    - Failure summary with LLM reasoning
    - All critical issues with LLM explanations (what, why, how to fix)
    - Fix roadmap with prioritized steps
    - Auto-fixable vs user-input-required categorization
    - Recommendations and resources
- ☐ JSON is pretty-printed (indented, human-readable)
- ☐ JSON validates against `CorrectionContext` schema (Appendix C)
- ☐ JSON filename: `<nwb_filename>_correction_context.json`
- ☐ JSON written to output directory alongside NWB file
- ☐ JSON path stored in global state for download/access
- ☐ Agent raises exception if serialization fails (defensive)

Priority: Critical

Epic 10: Web API Layer

Story 10.1: FastAPI Application Setup

Depends on: None (foundational)

As a developer **I want** a FastAPI application with CORS enabled **So that** React frontend can communicate with backend

Acceptance Criteria:

- ☐ FastAPI app initialized with title, description, version
- ☐ CORS middleware configured for localhost:3000
- ☐ Health check endpoint returns server status

- ☐ API info endpoint returns version and capabilities
- ☐ Static file serving configured for React build
- ☐ API documentation auto-generated at /docs

Priority: Critical

Story 10.2: File Upload Endpoint

Depends on: Story 10.1, Story 2.1

As a user **I want** to upload data files via web interface **So that** I can start conversion without CLI

Acceptance Criteria:

- ☐ Endpoint accepts multiple file uploads
- ☐ Endpoint accepts metadata form fields (subject_id, species, description, date)
- ☐ Endpoint validates all files uploaded successfully
- ☐ Endpoint checks if system is already processing (return 409 Conflict if busy)
- ☐ Endpoint saves files to upload directory
- ☐ Endpoint updates global state to processing
- ☐ Endpoint starts conversion in background task
- ☐ Endpoint returns 202 Accepted immediately
- ☐ Endpoint handles upload errors gracefully

Priority: Critical

Story 10.3: Background Task Processing

Depends on: Story 10.2, Story 1.2

As the system **I want** to process conversions asynchronously **So that** API remains responsive during long operations

Acceptance Criteria:

- ☐ Background task invokes conversion agent via MCP
- ☐ Task waits for conversion completion

- [ ] Task invokes evaluation agent with NWB file path
- [ ] Task updates global state at each stage
- [ ] Task handles agent errors gracefully
- [ ] Task marks global state as completed or failed
- [ ] Only one task runs at a time (single session constraint)

Priority: Critical

Story 10.4: Status API

Depends on: Story 2.1

As a user **I want** to query current conversion status **So that** I can see progress and results

Acceptance Criteria:

- [ ] GET /api/status returns global state
- [ ] Response includes status, validation_status, stages, metadata
- [ ] Response includes output paths if available
- [ ] Response includes current stage and progress
- [ ] Response includes logs
- [ ] Validation_status values: null (not yet validated), passed, passed_accepted, passed_improved, failed_user_declined, failed_user_abandoned
- [ ] Response includes validation details: issue counts by severity (CRITICAL, ERROR, WARNING, BEST_PRACTICE)
- [ ] Response time <100ms

Priority: High

Story 10.5: WebSocket Progress Streaming

Depends on: Story 2.1

As a user **I want** real-time progress updates **So that** I see conversion happening live

Acceptance Criteria:

- [ ] WebSocket endpoint at /ws
- [ ] Client receives progress updates as they occur
- [ ] Updates include stage, status, message
- [ ] Updates broadcast to all connected clients
- [ ] Connection closes when conversion completes
- [ ] Connection handles client disconnects gracefully
- [ ] Multiple clients can watch the same conversion

Priority: High

Story 10.6: Download Endpoints

Depends on: Story 6.4, Story 9.5, Story 9.6

As a user **I want** to download converted files and reports **So that** I can use the results

Acceptance Criteria:

- [ ] GET /api/download/nwb downloads current NWB file
- [ ] GET /api/download/report downloads PDF or JSON report
- [ ] Correct Content-Type headers set
- [ ] Proper filename in Content-Disposition
- [ ] 404 returned if file not found or conversion incomplete
- [ ] Large files stream efficiently

Priority: Critical

Story 10.7: Logs API

Depends on: Story 2.1

As a user **I want** to view conversion logs **So that** I can debug issues or understand what happened

Acceptance Criteria:

- [ ] GET /api/logs returns logs from global state
- [ ] Logs ordered chronologically

- ☐ Logs include timestamp, level, component, message
- ☐ Logs filterable by level (optional query param)
- ☐ Response time <200ms
- ☐ Returns last 500 log entries

Priority: Medium

Epic 11: React Web UI

Story 11.1: React Application Setup

Depends on: None (foundational)

As a developer **I want** a React application with TypeScript and Material-UI **So that** I can build a modern, type-safe UI

Acceptance Criteria:

- ☐ React app created with TypeScript template
- ☐ Material-UI (MUI) installed and configured
- ☐ Theme configured (colors, typography)
- ☐ Routing setup (if multi-page)
- ☐ API client service created
- ☐ Build process configured
- ☐ App runs on localhost:3000

Priority: Critical

Story 11.2: File Upload Component

Depends on: Story 11.1

As a user **I want** to drag-and-drop files or browse to upload **So that** I can easily provide my data

Acceptance Criteria:

- [ ] Component accepts drag-and-drop
- [ ] Component has file browse button
- [ ] Component shows file list after selection
- [ ] Component validates file selection (not empty)
- [ ] Component allows file removal before upload
- [ ] Component displays file sizes
- [ ] Component disables upload if system is busy (409 status check)
- [ ] Visual feedback for drag-over state

Priority: Critical

Story 11.3: Metadata Form

Depends on: Story 11.1

As a user **I want** to fill in required metadata in a form **So that** my NWB file has complete information

Acceptance Criteria:

- [ ] Form has fields for subject_id, species, description, session_date
- [ ] All fields have labels and helper text
- [ ] Species field has autocomplete suggestions (Mus musculus, Rattus norvegicus, Homo sapiens)
- [ ] Date field has date-time picker
- [ ] Required fields marked with asterisk
- [ ] Form validates before submission
- [ ] Clear validation error messages
- [ ] Form submits with files to API

Priority: Critical

Story 11.4: Progress View Component

Depends on: Story 11.1, Story 10.5

As a user **I want** to see real-time conversion progress **So that** I know the system is working and how long it will take

Acceptance Criteria:

- [ ] Component connects to WebSocket /ws on mount
- [ ] Component displays current stage (conversion, evaluation, report_generation)
- [ ] Component shows stage status indicators
- [ ] Component displays current operation message
- [ ] Component updates in real-time as messages arrive
- [ ] Component handles WebSocket disconnections
- [ ] Component shows completion state (success/failure)
- [ ] Component auto-refreshes results when complete

Priority: High

Story 11.5: Results Display Component

Depends on: Story 11.1, Story 10.4

As a user **I want** to see conversion results and download outputs **So that** I can access my converted files

Acceptance Criteria:

- [ ] Component fetches status from /api/status after completion
- [ ] Component displays validation status prominently with visual indicators:
    - PASSED (no issues): Green checkmark icon + "Perfect! No issues found."
    - PASSED_WITH_ISSUES: Yellow warning icon + "Valid with N warnings"
    - FAILED: Red X icon + "Validation failed with N errors"
- [ ] Component shows NWB file size and path
- [ ] Component shows validation summary with issue breakdown by severity:
    - CRITICAL: X issues
    - ERROR: Y issues
    - WARNING: Z issues
    - BEST_PRACTICE: W suggestions
- [ ] Component has download buttons appropriate for validation status:

    - PASSED/PASSED_WITH_ISSUES: "Download NWB" + "Download PDF Report"
    - FAILED: "Download NWB (with errors)" + "Download JSON Report"
- [ ] Component displays context-appropriate success messages:
    - PASSED: "Your file is perfect and ready to use!"
    - PASSED_WITH_ISSUES (accepted): "Your file is valid and ready to use. Some minor improvements were suggested."
    - PASSED (after improvement): "Success! All warnings resolved. Your file is now perfect!"
- [ ] Component displays helpful message for FAILED with actionable next steps
- [ ] Download buttons trigger file downloads with correct MIME types

Priority: Critical

Story 11.6: Log Viewer Component

Depends on: Story 11.1, Story 10.7

As a user **I want** to view conversion logs **So that** I can understand what happened or debug issues

Acceptance Criteria:

- [ ] Component fetches logs from /api/logs
- [ ] Logs displayed in reverse chronological order (newest first)
- [ ] Each log entry shows timestamp, level, component, message
- [ ] Log levels color-coded (ERROR red, WARNING yellow, INFO blue)
- [ ] Component has refresh button
- [ ] Component auto-scrolls to newest logs
- [ ] Component handles empty logs gracefully

Priority: Medium

Story 11.7: Basic Error Handling in UI

Depends on: Story 11.1

As a user **I want** to see error messages when things go wrong **So that** I understand what happened

Acceptance Criteria:

- [ ] Upload failures show error message from API
- [ ] 409 Conflict shows "System is busy processing another conversion"
- [ ] Network errors show "Connection error - check your internet"
- [ ] API errors display error message in UI
- [ ] Form validation errors highlight problematic fields
- [ ] Errors shown in toast/alert component
- [ ] Errors are dismissible
- [ ] Full error details logged to browser console

Priority: Medium

Epic 12: Integration & Polish

Story 12.1: End-to-End Integration Test

Depends on: All previous stories

As a developer **I want** an automated test that verifies the complete pipeline **So that** I know the system works end-to-end

Acceptance Criteria:

- [ ] Test registers agents with MCP server
- [ ] Test initializes global state
- [ ] Test invokes conversion agent with sample data
- [ ] Test verifies NWB file created
- [ ] Test invokes evaluation agent
- [ ] Test verifies report generated for all validation statuses:
    - PASSED (no issues): PDF report generated
    - PASSED_WITH_ISSUES: PDF with warnings, user accepts as-is, validation_status="passed_accepted"

    - PASSED_WITH_ISSUES: PDF with warnings, user improves file, validation_status="passed_improved"
    - FAILED: JSON report generated, user declines retry, validation_status="failed_user_declined"
- ☐ Test checks global state is completed with correct validation_status
- ☐ Test verifies all three validation paths (PASSED, PASSED_WITH_ISSUES, FAILED)
- ☐ Test verifies file versioning (v1, v2, v3) with SHA256 checksums
- ☐ Test completes in <5 minutes

Priority: Critical

Story 12.2: Sample Dataset Creation

Depends on: None (foundational)

As a developer **I want** minimal test datasets for development and testing **So that** I don't need access to real large files

Acceptance Criteria:

- ☐ Script creates one minimal toy dataset in a directory (e.g., SpikeGLX format)
- ☐ Dataset: 10 seconds, 32-64 channels (small for fast testing)
- ☐ Dataset size <10 MB (toy data for integration test timeout validation)
- ☐ Dataset is valid and convertible by NeuroConv
- ☐ Dataset deliberately has quality issues (e.g., missing recommended metadata) to test correction loop
- ☐ Script is simple and documented
- ☐ Dataset committed to repository (in tests/fixtures/ or similar)

Priority: Critical

Story 12.3: Installation Script

Depends on: None (foundational)

As a user **I want** an automated installation script **So that** setup is easy and error-free

Acceptance Criteria:

- [ ] Script installs system dependencies
- [ ] Script sets up Python environment via Pixi
- [ ] Script creates necessary directories (uploads, outputs)
- [ ] Script verifies installation
- [ ] Script provides next steps
- [ ] Script works on Linux and macOS

Priority: High

Story 12.4: Quick Start Script

Depends on: Story 12.3

As a user **I want** a quick start script that demonstrates the system **So that** I can verify installation and see it working

Acceptance Criteria:

- [ ] Script creates sample data if not present
- [ ] Script starts backend server
- [ ] Script starts frontend server
- [ ] Script provides URL to access UI
- [ ] Script shows example CLI command
- [ ] Script cleans up on exit
- [ ] Script documented in README

Priority: Medium

Story 12.5: Error Recovery Testing

Depends on: Story 12.1

As a developer **I want** to test error scenarios **So that** the system handles failures gracefully

Acceptance Criteria:

- [ ] Test invalid file format handling
- [ ] Test network errors (API unreachable)
- [ ] Test LLM API failures
- [ ] Test concurrent upload attempts (409 Conflict)
- [ ] Test disk full scenarios
- [ ] Test large file handling
- [ ] All errors logged appropriately
- [ ] User sees helpful error messages

Priority: High

Story 12.6: Integration Test Timeouts

Depends on: Story 12.1

As a developer **I want** integration tests with realistic timeouts on toy datasets **So that** I can detect context engineering problems early

Acceptance Criteria:

- [ ] End-to-end integration test uses toy dataset (<10 MB)
- [ ] Complete pipeline (scan → detect → convert → evaluate → report) has reasonable timeout (5-10 minutes)
- [ ] Test fails with timeout error if agents are "stuck" (indicates context engineering issue)
- [ ] Test measures actual duration and logs it
- [ ] Test verifies single-session constraint (concurrent attempts fail immediately)
- [ ] No performance tests on large files (not assessable in MVP)
- [ ] Timeout is generous to allow for LLM API latency and agent communication

Priority: High

System Requirements

Functional Requirements

Core Capabilities:

- [ ] Accept directory upload containing neurophysiology data via web interface
- [ ] Convert any NeuroConv-supported format to NWB (delegates format detection to NeuroConv)
- [ ] Validate schema compliance (PyNWB read test)
- [ ] Evaluate quality using NWB Inspector (metadata completeness, best practices)
- [ ] Generate LLM-enhanced evaluation reports (PDF for PASSED/PASSED_WITH_ISSUES, JSON for FAILED)
- [ ] Provide structured logs for provenance tracking
- [ ] Provide web UI for file upload, progress monitoring, and downloads
- [ ] Provide REST API for programmatic access
- [ ] Track single-session state in memory

User Experience:

- [ ] Users upload files/directories via drag-and-drop in web UI
- [ ] Users fill in required metadata through web form
- [ ] Users see real-time conversion progress via WebSocket updates
- [ ] Users download NWB files and reports via web UI
- [ ] Users view detailed logs in web interface
- [ ] Users receive clear feedback on validation status (PASSED/PASSED_WITH_ISSUES/FAILED)

System Behavior:

- [ ] MCP server routes messages between agents
- [ ] Agents communicate via standardized protocol
- [ ] Global state tracks single conversion in memory
- [ ] LLM failures don't crash the system
- [ ] All actions logged for auditing

Non-Functional Requirements

Performance (MVP - "Good Enough to Work"):

- [ ] Integration tests with **toy dataset** (≤10 MB, simple SpikeGLX recording) complete in **≤10 minutes** (generous timeout for LLM latency)

- [ ] No memory/CPU/disk limits enforced—system uses whatever resources available (optimize post-MVP)
- [ ] No performance benchmarks for large files in MVP

Reliability (Defensive Error Handling):

- [ ] System raises exceptions **immediately** when something is wrong (no silent failures, no default values that hide problems)
- [ ] All exceptions include **full diagnostic context** in structured JSON format:

  ```
  {
    "timestamp": "ISO 8601 timestamp",
    "component": "agent_name or service_name",
    "error_code": "unique_error_identifier",
    "message": "human-readable error description",
    "stack_trace": "full Python traceback",
    "state_snapshot": {
      "session_id": "...",
      "current_stage": "...",
      "input_files": [...],
      "metadata": {...}
    }
  }
  ```

- [ ] Failed conversions preserve all logs (saved to `logs/{session_id}/` directory) before raising exception
- [ ] No graceful error handling or automatic retry logic (except user-controlled correction loop in Story 8.7)
- [ ] **LLM Error Handling Strategy**:
  - **Critical LLM failures** (Stories 4.4, 9.3, 9.4): Raise `LLMAPIException` with HTTP status code, API error message, retry-after header. System stops correction loop.
  - **Optional LLM failures** (Story 5.3 format detection): Log warning, degrade gracefully to NeuroConv default. No exception raised.
- [ ] File I/O errors raise `FileProcessingException` with file path, operation attempted, and OS error code
- [ ] Schema validation failures raise `NWBValidationException` with PyNWB error details and line numbers
- [ ] All agent communication errors include MCP message ID, sender, receiver, and payload

Scalability:

- ☐ **Single conversion at a time** (MVP constraint)—concurrent uploads blocked with simple error message
- ☐ Global state is **in-memory** (Python dict)—no database needed
- ☐ File system handles uploaded files (no enforced size limit in MVP)

Usability (MVP - Basic Functionality):

- ☐ Web UI allows file upload, shows progress, provides download links
- ☐ Progress updates via WebSocket (no latency requirement—just working updates)
- ☐ Error messages shown in UI (simple text display—fancy formatting optional)
- ☐ Validation status displayed with basic indicators (text or simple colored badges)

Maintainability:

- ☐ Code coverage **≥80%** (measured by pytest-cov, excluding MCP boilerplate)
- ☐ All agents are **independent Python modules** (no direct imports between agents, only MCP communication)
- ☐ Logging uses **structured JSON format** (JSON Lines `.jsonl` files) with fields: `timestamp`, `level`, `component`, `event`, `data`
- ☐ All configuration via **environment variables** (`.env` file for local, system env for deployment):
  - `ANTHROPIC_API_KEY`: Required, no default
  - `UPLOAD_DIR`: Default `./uploads`
  - `OUTPUT_DIR`: Default `./outputs`
  - `LOG_DIR`: Default `./logs`
  - `MAX_UPLOAD_SIZE_GB`: Default `100`
- ☐ Format support updates require **only NeuroConv version bump** (no code changes in agents)—tested by upgrading NeuroConv in isolated test

Security (MVP - Basic Safety):

- ☐ API keys in environment variables (never hardcoded)
- ☐ File upload has **reasonable size limit** (e.g., 50 GB—prevent system crashes from huge uploads)
- ☐ Basic path validation: reject paths with `..` (prevent directory traversal)
- ☐ No authentication in MVP (local deployment only)

Notes:

- ✅ **Priority**: Make it work first, optimize later

- ✅ **"Good enough"**: Basic error handling, simple logging, minimal validation
- ✅ **Defer optimization**: Memory limits, fancy error schemas, test coverage >50%, multi-browser testing—all post-MVP

Dependencies

External Services

- **Anthropic Claude API**: For LLM-powered analysis and reports
  - Requires API key (ANTHROPIC_API_KEY environment variable)
  - Rate limits apply
  - **Required**: System throws errors if LLM unavailable (no fallback)

Python Libraries

- **NeuroConv** (≥0.4.0): Data format conversion and auto-detection
- **PyNWB** (≥2.6.0): NWB file handling and schema validation
- **NWB Inspector** (≥0.4.30): Quality evaluation
- **FastAPI**: Web framework for REST API and WebSocket support
- **Uvicorn**: ASGI server for FastAPI
- **Anthropic SDK** (≥0.18.0): Claude API client for LLM analysis
- **Pydantic** (≥2.0): Type-safe data schemas (MCP messages, global state)
- **ReportLab** (≥3.6.0) or **Quarto**: PDF report generation (Quarto recommended to avoid vendor lock-in)

Frontend Libraries

- **React** (18+): UI framework
- **TypeScript**: Type safety
- **Material-UI (MUI)** (≥5.0): Component library with pre-built UI elements
  - `@mui/material`: Core components (Button, TextField, Card, etc.)
  - `@mui/icons-material`: Icon library (CheckCircle, Warning, Error, etc.)
  - `@emotion/react` + `@emotion/styled`: Required peer dependencies for MUI

- **Axios**: HTTP client for API communication
- **React-Dropzone**: File upload with drag-and-drop

Infrastructure

- **Pixi**: Python environment management

Success Criteria

MVP is DONE when:

Core Three-Agent Loop Works:

1. ✅ User uploads directory via web UI
2. ✅ User fills in required metadata via web form
3. ✅ Conversion Agent detects format via NeuroConv and converts to NWB
4. ✅ Evaluation Agent validates schema (PyNWB) and evaluates quality (Inspector)
5. ✅ Conversation Agent orchestrates user-controlled correction loop
6. ✅ LLM analyzes evaluation results and generates actionable reports (PDF/JSON)
7. ✅ Self-correction loop completes (user approves retry → reconvert → re-evaluate)
8. ✅ User sees real-time progress via WebSocket updates in UI
9. ✅ User downloads NWB file and report via web UI

Quality Standards:

1. ✅ End-to-end integration test passes with toy dataset (<2 min)
2. ✅ All agent interactions use MCP protocol
3. ✅ System raises defensive errors (no silent failures)
4. ✅ Structured logs provide complete provenance trail
5. ✅ Sample toy dataset available for testing

Deliverables:

1. ✅ Three-agent system (Conversation, Conversion, Evaluation)
2. ✅ MCP server with message routing
3. ✅ Web UI (React + TypeScript + Tailwind CSS)
4. ✅ FastAPI backend with WebSocket support

5. ✅ Integration tests with timeouts
6. ✅ Pixi environment configuration
7. ✅ Sample toy dataset for testing

Explicitly NOT Required for MVP:

- README documentation (add later)
- Performance optimization for large files
- Deployment/containerization (local development only)
- CLI interface (web UI is sufficient)

Appendix A: API Endpoint Summary

```
Authentication: None (future enhancement)
Base URL: http://localhost:8080

Endpoints (Single Session):
POST   /api/upload                     Upload files + metadata (409 if busy)
GET    /api/status                     Get current conversion status
(includes correction_attempt, awaiting_retry_approval)
GET    /api/logs                       Get conversion logs
POST   /api/retry-approval             User approves or declines retry
attempt
GET    /api/correction-context         Get validation failure summary for
retry decision
POST   /api/user-input                 Submit user input for correction
GET    /api/download/nwb               Download NWB file (latest version)
GET    /api/download/nwb/v{N}          Download specific NWB version
GET    /api/download/report            Download report (works for PASSED or
FAILED status)
WS     /ws                             WebSocket progress updates (includes
retry approval stages)
GET    /health                         Health check
GET    /api/info                       API information
```

Appendix B: Data Schemas

This appendix defines the exact data structures referenced throughout the user stories. These schemas ensure type safety and consistency across all implementation.

MCP Message Schema

All inter-agent messages use this standardized structure for communication via the MCP server.

```
from pydantic import BaseModel, Field
from datetime import datetime
from typing import Any, Dict, Optional
from uuid import uuid4

class MCPMessage(BaseModel):
    """
    Standard message format for Model Context Protocol communication.
    Used by all agents to communicate via the MCP server.
    """
    message_id: str = Field(default_factory=lambda: str(uuid4()))
    target_agent: str = Field(..., description="Target agent name from
registry (e.g., 'conversation_agent', 'conversion_agent',
'evaluation_agent')")
    action: str = Field(..., description="MCP tool/method name to invoke
(e.g., 'validate_metadata', 'convert_file', 'generate_report')")
    context: Dict[str, Any] = Field(default_factory=dict,
description="Request-specific parameters and data")
    timestamp: datetime = Field(default_factory=datetime.now)
    source_agent: Optional[str] = Field(None, description="Optional: which
agent sent this message")
    correlation_id: Optional[str] = Field(None, description="Optional: for
tracing related messages")

    class Config:
        json_schema_extra = {
            "example": {
                "message_id": "abc-123-def-456",
                "target_agent": "conversion_agent",
                "action": "convert_file",
                "context": {
                    "input_path": "/uploads/spikeglx_data.bin",
                    "metadata": {"subject_id": "mouse_001"}
                },
                "timestamp": "2025-10-14T10:30:00Z",
                "source_agent": "conversation_agent"
            }
        }
```

Global State Schema

Single global state object tracking the current conversion session (Stories 2.1, 2.2).

```
from pydantic import BaseModel, Field
from enum import Enum
from datetime import datetime
from typing import List, Dict, Any, Optional

class ConversionStatus(str, Enum):
    """Overall conversion status"""
    IDLE = "idle"
    PROCESSING = "processing"
    COMPLETED = "completed"
    FAILED = "failed"

class ValidationStatus(str, Enum):
    """Granular validation outcome status"""
    PASSED = "passed"  # No issues at all
    PASSED_ACCEPTED = "passed_accepted"  # User accepted file with warnings
    PASSED_IMPROVED = "passed_improved"  # Warnings resolved through
improvement
    FAILED_USER_DECLINED = "failed_user_declined"  # User declined retry
    FAILED_USER_ABANDONED = "failed_user_abandoned"  # User cancelled during
input

class StageStatus(str, Enum):
    """Status of individual pipeline stages"""
    PENDING = "pending"
    IN_PROGRESS = "in_progress"
    COMPLETED = "completed"
    FAILED = "failed"

class LogLevel(str, Enum):
    """Log severity levels"""
    DEBUG = "DEBUG"
    INFO = "INFO"
    WARNING = "WARNING"
    ERROR = "ERROR"
    CRITICAL = "CRITICAL"

class LogEntry(BaseModel):
    """Individual log entry"""
    timestamp: datetime
    level: LogLevel
    component: str = Field(..., description="Component name (e.g.,
'mcp_server', 'conversation_agent')")
    message: str
    metadata: Dict[str, Any] = Field(default_factory=dict)

class Stage(BaseModel):
    """Pipeline stage tracking"""
    name: str = Field(..., description="Stage name: 'conversion',
'evaluation', 'report_generation'")
    status: StageStatus
    start_time: Optional[datetime] = None
    end_time: Optional[datetime] = None
    output_path: Optional[str] = None
    error_message: Optional[str] = None
    metadata: Dict[str, Any] = Field(default_factory=dict)
```

```python
class GlobalState(BaseModel):
    """
    Single global state object for the current conversion session.
    Tracks all aspects of the conversion pipeline.
    """
    status: ConversionStatus
    validation_status: Optional[ValidationStatus] = None
    input_path: Optional[str] = None
    output_path: Optional[str] = None
    metadata: Dict[str, Any] = Field(default_factory=dict, description="NWB
metadata fields (subject_id, species, session_description, etc.)")
    logs: List[LogEntry] = Field(default_factory=list)
    stages: List[Stage] = Field(default_factory=list)
    timestamps: Dict[str, datetime] = Field(default_factory=dict,
description="Key: event name, Value: timestamp (e.g., 'upload',
'conversion_start')")
    correction_attempt: int = Field(default=0, description="Number of
correction attempts (0 = first attempt)")

    class Config:
        json_schema_extra = {
            "example": {
                "status": "processing",
                "validation_status": None,
                "input_path": "/uploads/spikeglx_data.bin",
                "output_path": "/outputs/mouse_001.nwb",
                "metadata": {
                    "subject_id": "mouse_001",
                    "species": "Mus musculus",
                    "session_description": "Neuropixels recording",
                    "session_start_time": "2025-10-14T09:00:00Z"
                },
                "logs": [],
                "stages": [
                    {
                        "name": "conversion",
                        "status": "completed",
                        "start_time": "2025-10-14T10:00:00Z",
                        "end_time": "2025-10-14T10:05:00Z"
                    }
                ],
                "timestamps": {
                    "upload": "2025-10-14T09:55:00Z",
                    "conversion_start": "2025-10-14T10:00:00Z"
                },
                "correction_attempt": 0
            }
        }
```

Validation Result Schema

Structure for NWB Inspector validation results (Stories 7.1, 7.2, 7.3).

```
from pydantic import BaseModel, Field
from enum import Enum
from datetime import datetime
from typing import List, Dict, Any

class IssueSeverity(str, Enum):
    """NWB Inspector issue severity levels"""
    CRITICAL = "CRITICAL"
    ERROR = "ERROR"
    WARNING = "WARNING"
    BEST_PRACTICE = "BEST_PRACTICE"

class OverallStatus(str, Enum):
    """Three-tier validation status"""
    PASSED = "PASSED"  # No issues at all
    PASSED_WITH_ISSUES = "PASSED_WITH_ISSUES"  # Only WARNING or
BEST_PRACTICE issues
    FAILED = "FAILED"  # Has CRITICAL or ERROR issues

class ValidationIssue(BaseModel):
    """Individual validation issue from NWB Inspector"""
    check_name: str = Field(..., description="Name of the NWB Inspector
check")
    severity: IssueSeverity
    message: str = Field(..., description="Human-readable description of the
issue")
    location: str = Field(..., description="Path in NWB file where issue
occurs (e.g., '/general/subject')")
    file_path: str = Field(..., description="Path to the NWB file")
    importance: Optional[str] = Field(None, description="NWB Inspector
importance level")

class FileInfo(BaseModel):
    """Comprehensive NWB file information (Story 7.1)"""
    nwb_version: str
    creation_date: datetime
    identifier: str
    session_description: str
    subject_id: Optional[str] = None
    species: Optional[str] = None
    age: Optional[str] = None
    sex: Optional[str] = None
    experimenter: Optional[List[str]] = None
    institution: Optional[str] = None
    lab: Optional[str] = None
    devices: List[str] = Field(default_factory=list)
    electrode_groups: List[str] = Field(default_factory=list)
    acquisition_data: List[Dict[str, Any]] = Field(default_factory=list)
    processing_modules: List[str] = Field(default_factory=list)
    file_size_bytes: int
    temporal_coverage_seconds: Optional[float] = None

class ValidationResult(BaseModel):
    """
    Complete validation result from Evaluation Agent.
    Passed between Evaluation Agent and Conversation Agent.
    """
    overall_status: OverallStatus
    issues: List[ValidationIssue] = Field(default_factory=list)
    issue_counts: Dict[IssueSeverity, int] = Field(
        default_factory=lambda: {
            IssueSeverity.CRITICAL: 0,
            IssueSeverity.ERROR: 0,
            IssueSeverity.WARNING: 0,
            IssueSeverity.BEST_PRACTICE: 0
        }
    )
    file_info: FileInfo
    timestamp: datetime = Field(default_factory=datetime.now)
    nwb_file_path: str
    checksum_sha256: Optional[str] = Field(None, description="SHA256
checksum of NWB file (Story 8.6)")

    class Config:
        json_schema_extra = {
            "example": {
                "overall_status": "PASSED_WITH_ISSUES",
                "issues": [
                    {
                        "check_name": "subject_age_check",
                        "severity": "WARNING",
                        "message": "Subject age is missing. Recommended for
DANDI archive.",
                        "location": "/general/subject",
                        "file_path": "/outputs/mouse_001.nwb"
                    }
                ],
                "issue_counts": {
                    "CRITICAL": 0,
                    "ERROR": 0,
                    "WARNING": 1,
                    "BEST_PRACTICE": 0
                },
                "file_info": {
                    "nwb_version": "2.6.0",
                    "subject_id": "mouse_001",
                    "species": "Mus musculus"
                },
                "timestamp": "2025-10-14T10:10:00Z",
                "nwb_file_path": "/outputs/mouse_001.nwb"
            }
        }
```

Correction Context Schema

Context passed from Evaluation Agent to Conversation Agent for corrections (Stories 8.1, 8.2, 8.3).

```
from pydantic import BaseModel, Field
from typing import List, Dict, Optional

class FixStrategy(BaseModel):
    """Suggested fix for an issue"""
    issue_id: str = Field(..., description="Reference to ValidationIssue")
    strategy: str = Field(..., description="Human-readable fix strategy")
    auto_fixable: bool = Field(..., description="Can system fix
automatically?")
    user_input_required: bool = Field(..., description="Does this require
user input?")
    user_prompt: Optional[str] = Field(None, description="If user input
required, what to ask")
    estimated_effort: Optional[str] = Field(None, description="'easy',
'medium', 'hard'")

class CorrectionContext(BaseModel):
    """
    Context passed from Evaluation Agent to Conversation Agent
    when validation fails or passes with issues.
    """
    validation_result: ValidationResult
    auto_fixable_issues: List[ValidationIssue] = Field(
        default_factory=list,
        description="Issues the system can fix automatically (Story 8.4)"
    )
    user_input_required_issues: List[ValidationIssue] = Field(
        default_factory=list,
        description="Issues requiring user to provide data (Story 8.5)"
    )
    suggested_fixes: List[FixStrategy] = Field(default_factory=list)
    attempt_number: int = Field(default=1, description="Which correction
attempt (1, 2, 3, ...)")
    previous_issues: Optional[List[ValidationIssue]] = Field(
        None,
        description="Issues from previous attempt (for detecting 'no
progress')"
    )
    llm_analysis: Optional[str] = Field(
        None,
        description="LLM's analysis of the issues (Story 4.4)"
    )

    class Config:
        json_schema_extra = {
            "example": {
                "validation_result": {"overall_status": "FAILED", "issues":
[...]},
                "auto_fixable_issues": [],
                "user_input_required_issues": [
                    {
                        "check_name": "subject_id_missing",
                        "severity": "ERROR",
                        "message": "Subject ID is required"
                    }
                ],
                "suggested_fixes": [
                    {
                        "issue_id": "subject_id_missing",
                        "strategy": "Prompt user to provide subject_id",
                        "auto_fixable": False,
                        "user_input_required": True,
                        "user_prompt": "What is the subject ID? (e.g.,
'mouse_001')"
                    }
                ],
                "attempt_number": 1
            }
        }
```

API Request/Response Schemas

Common schemas for the FastAPI endpoints (Epic 10).

```python
from pydantic import BaseModel, Field
from typing import List, Optional

class UploadRequest(BaseModel):
    """Request body for file upload (Story 10.2)"""
    subject_id: str
    species: str
    session_description: str
    session_start_time: str = Field(..., description="ISO 8601 format")
    experimenter: Optional[str] = None
    institution: Optional[str] = None
    lab: Optional[str] = None
    age: Optional[str] = None
    sex: Optional[str] = None
    weight: Optional[str] = None

class StatusResponse(BaseModel):
    """Response from GET /api/status (Story 10.4)"""
    status: ConversionStatus
    validation_status: Optional[ValidationStatus]
    current_stage: Optional[Stage]
    stages: List[Stage]
    metadata: Dict[str, Any]
    logs: List[LogEntry]
    validation_details: Optional[Dict[IssueSeverity, int]] = None
    output_path: Optional[str] = None
    error_message: Optional[str] = None

class RetryApprovalRequest(BaseModel):
    """Request body for POST /api/retry-approval"""
    approved: bool = Field(..., description="True = user approves retry,
False = user declines")
    accept_as_is: Optional[bool] = Field(False, description="For
PASSED_WITH_ISSUES: accept file without improvement")

class UserInputRequest(BaseModel):
    """Request body for POST /api/user-input"""
    field_name: str
    value: Any

class WebSocketMessage(BaseModel):
    """WebSocket progress update message (Story 10.5)"""
    type: str = Field(..., description="Message type: 'progress',
'stage_update', 'notification', 'error'")
    message: str
    stage: Optional[str] = None
    status: Optional[str] = None
    timestamp: datetime = Field(default_factory=datetime.now)
    metadata: Dict[str, Any] = Field(default_factory=dict)
```

Usage Examples

Example 1: MCP Message Flow

```
# Story 4.2 → Story 6.1: Conversation Agent sends metadata to Conversion
Agent
message = MCPMessage(
    target_agent="conversion_agent",
    action="collect_metadata",
    context={
        "subject_id": "mouse_001",
        "species": "Mus musculus",
        "session_description": "Neuropixels recording session 1",
        "session_start_time": "2025-10-14T09:00:00Z"
    },
    source_agent="conversation_agent"
)

# MCP server routes message to conversion_agent
response = mcp_server.route_message(message)
```

Example 2: Validation Result Processing

```
# Story 7.3: Evaluation Agent creates validation result
validation_result = ValidationResult(
    overall_status=OverallStatus.PASSED_WITH_ISSUES,
    issues=[
        ValidationIssue(
            check_name="subject_age_missing",
            severity=IssueSeverity.WARNING,
            message="Subject age is not specified",
            location="/general/subject",
            file_path="/outputs/mouse_001.nwb"
        )
    ],
    issue_counts={
        IssueSeverity.CRITICAL: 0,
        IssueSeverity.ERROR: 0,
        IssueSeverity.WARNING: 1,
        IssueSeverity.BEST_PRACTICE: 0
    },
    file_info=file_info,
    nwb_file_path="/outputs/mouse_001.nwb"
)

# Story 8.1: Generate correction context
context = CorrectionContext(
    validation_result=validation_result,
    auto_fixable_issues=[],  # Age can't be auto-inferred
    user_input_required_issues=[validation_result.issues[0]],
    suggested_fixes=[
        FixStrategy(
            issue_id="subject_age_missing",
            strategy="Prompt user to provide subject age",
            auto_fixable=False,
            user_input_required=True,
            user_prompt="What is the subject's age? (e.g., 'P90D' for 90
days)"
        )
    ],
    attempt_number=1
)
```

Example 3: Global State Updates

```
# Story 2.1: Initialize global state
global_state = GlobalState(
    status=ConversionStatus.IDLE,
    validation_status=None,
    timestamps={"system_start": datetime.now()}
)

# Story 2.2: Update stage tracking
conversion_stage = Stage(
    name="conversion",
    status=StageStatus.IN_PROGRESS,
    start_time=datetime.now(),
    metadata={"format": "SpikeGLX", "confidence": "high"}  # or Intan,
OpenEphys, etc.
)
```

```
    global_state.stages.append(conversion_stage)
    global_state.status = ConversionStatus.PROCESSING

    # Story 7.3: Update validation status
    conversion_stage.status = StageStatus.COMPLETED
    conversion_stage.end_time = datetime.now()
    global_state.validation_status = ValidationStatus.PASSED_IMPROVED
```

End of Document