Multi-Agent NWB Conversion Pipeline - Technical Specification

Executive Summary

This document provides complete technical specifications for implementing a multi-agent system for converting heterogeneous neuroscience data into the Neurodata Without Borders (NWB) format using the Model Context Protocol (MCP) as the core communication framework.

---

1. System Architecture Overview

1.1 Core Technology Stack

- **Agent Framework**: MCP (Model Context Protocol) for inter-agent communication
- **Runtime Environment**: Python 3.10+
- **Orchestration**: Apache Airflow or Prefect for workflow management
- **Message Queue**: RabbitMQ or Redis for async communication
- **Data Storage**: MinIO for object storage, PostgreSQL for metadata
- **Containerization**: Docker & Kubernetes for deployment

1.2 MCP Implementation Strategy

python

```
# MCP Server Configuration Structure
{
    "protocol_version": "1.0",
    "server_info": {
        "name": "nwb-conversion-pipeline",
        "version": "0.1.0",
        "capabilities": {
            "tools": true,
            "resources": true,
            "prompts": true
        }
    },
    "agents": [
        "conversation_agent",
        "generation_agent",
        "evaluation_agent",
        "tuie_agent"
    ]
}
```

2. Agent Specifications

2.1 Conversation Agent

Purpose

Entry point for user interaction, metadata collection, and requirement gathering.

MCP Implementation

```python
class ConversationAgentMCP:
    def __init__(self):
        self.mcp_server = MCPServer(
            name="conversation_agent",
            version="1.0.0"
        )
        self.tools = [
            "gather_dataset_info",
            "extract_metadata",
            "validate_user_input",
            "query_knowledge_graph"
        ]

    async def handle_request(self, request: MCPRequest):
        # Request structure
        {
            "method": "conversation/initiate",
            "params": {
                "user_id": str,
                "dataset_path": str,
                "metadata": dict,
                "missing_fields": list
            }
        }
```

API Endpoints

- `POST /conversation/start` - Initialize conversation session
- `POST /conversation/metadata` - Submit metadata
- `GET /conversation/status/{session_id}` - Get conversation status
- `POST /conversation/validate` - Validate collected information

Data Structures

```python
@dataclass
class ConversationSession:
    session_id: str
    user_id: str
    timestamp: datetime
    dataset_info: DatasetInfo
    metadata_status: MetadataStatus
    missing_fields: List[str]
    ai_suggestions: List[Suggestion]

@dataclass
class DatasetInfo:
    path: str
    format: str
    size_bytes: int
    file_count: int
    data_types: List[str]
    sampling_rate: Optional[float]
    channels: Optional[int]
```

Integration Points

- **Knowledge Graph**: Query for existing metadata patterns
- **LinkML Schema**: Validate metadata against schemas
- **User Interface**: WebSocket for real-time interaction

2.2 Generation Agent (Conversion Agent)

Purpose

Execute the actual data conversion using NeuroConv, manage transformation pipeline.

MCP Implementation

```python
```

```
class GenerationAgentMCP:
    def __init__(self):
        self.mcp_server = MCPServer(
            name="generation_agent",
            version="1.0.0"
        )
        self.neuroconv = NeuroConvInterface()

    async def convert_dataset(self, request: MCPRequest):
        # Request structure
        {
            "method": "conversion/execute",
            "params": {
                "session_id": str,
                "input_path": str,
                "output_path": str,
                "conversion_config": dict,
                "metadata": dict,
                "provenance": dict
            }
        }

        # Response structure
        {
            "status": "success|processing|failed",
            "nwb_file_path": str,
            "conversion_log": str,
            "warnings": list,
            "provenance_record": dict
        }
```

Conversion Pipeline Steps

1. **Data Interface Selection**

   python

   ```
   def select_data_interface(data_format: str) -> DataInterface:
       interfaces = {
           "blackrock": BlackrockInterface,
           "intan": IntanInterface,
           "openephys": OpenEphysInterface,
           "spikeglx": SpikeGLXInterface
       }
       return interfaces.get(data_format)
   ```

2. **Metadata Injection**

python

```
def inject_metadata(nwbfile: NWBFile, metadata: dict):
    # Add subject information
    # Add experiment metadata
    # Add device information
    # Add electrode configuration
    # Mark AI-suggested fields
```

3. **Provenance Tracking**

python

```
@dataclass
class ProvenanceRecord:
    conversion_id: str
    timestamp: datetime
    source_files: List[str]
    conversion_parameters: dict
    tool_versions: dict
    user_modifications: List[Modification]
    ai_suggestions_applied: List[str]
```

2.3 Evaluation Agent

Purpose

Validate generated NWB files for compliance and quality.

MCP Implementation

python

```
class EvaluationAgentMCP:
    def __init__(self):
        self.mcp_server = MCPServer(
            name="evaluation_agent",
            version="1.0.0"
        )
        self.inspector = NWBInspector()

    async def validate_nwb(self, request: MCPRequest):
        # Request structure
        {
            "method": "validation/execute",
            "params": {
                "nwb_file_path": str,
                "validation_level": "basic|comprehensive|custom",
                "custom_checks": list
            }
        }
```

Validation Checks

python

```
class ValidationSuite:
    def __init__(self):
        self.checks = [
            SchemaComplianceCheck(),
            BestPracticesCheck(),
            MetadataCompletenessCheck(),
            DataIntegrityCheck(),
            TimeSeriesValidation(),
            UnitConsistencyCheck()
        ]

    async def run_validation(self, nwb_file: str) -> ValidationReport:
        results = []
        for check in self.checks:
            result = await check.validate(nwb_file)
            results.append(result)
        return ValidationReport(results)
```

Error Classification

python

```
class ErrorSeverity(Enum):
    CRITICAL = "critical"  # Must fix before proceeding
    WARNING = "warning"    # Should fix but not blocking
    INFO = "info"          # Suggestions for improvement

@dataclass
class ValidationError:
    severity: ErrorSeverity
    location: str
    message: str
    suggested_fix: Optional[str]
    auto_fixable: bool
```

2.4 Tool Use Instance Evaluator (TUIE)

Purpose

Meta-validator ensuring correct and efficient tool usage across the pipeline.

MCP Implementation

python

```python
class TUIEAgentMCP:
    def __init__(self):
        self.mcp_server = MCPServer(
            name="tuie_agent",
            version="1.0.0"
        )
        self.performance_metrics = {}

    async def evaluate_tool_usage(self, request: MCPRequest):
        # Monitor tool invocations
        # Analyze performance metrics
        # Suggest optimizations
        # Detect redundant operations
```

Monitoring Metrics

python

```
@dataclass
class ToolUsageMetrics:
    tool_name: str
    invocation_count: int
    average_duration: float
    error_rate: float
    resource_usage: ResourceMetrics
    optimization_suggestions: List[str]

@dataclass
class ResourceMetrics:
    cpu_usage: float
    memory_usage: float
    disk_io: float
    network_io: float
```

---

3. Intermediate Processes Specifications

3.1 Data Preprocessing and Analysis

Implementation

python

```
class DataPreprocessor:
    def __init__(self):
        self.supported_formats = [
            'dat', 'bin', 'continuous', 'nsx', 'nev',
            'rhd', 'rhs', 'oebin', 'kwik'
        ]

    async def preprocess(self, input_data: InputData) -> ProcessedData:
        # 1. Format detection
        format_type = self.detect_format(input_data)

        # 2. Data validation
        validation_result = self.validate_data_integrity(input_data)

        # 3. Standardization
        standardized = self.standardize_format(input_data, format_type)

        # 4. Quality checks
        quality_report = self.run_quality_checks(standardized)

        return ProcessedData(
            data=standardized,
            format=format_type,
            quality_report=quality_report
        )
```

Data Standardization Pipeline

python

```
class StandardizationPipeline:
    stages = [
        SamplingRateNormalization(),
        ChannelReordering(),
        UnitConversion(),
        TimeAlignement(),
        FilteringOptional(),
        ArtifactDetection()
    ]

    async def process(self, data: np.ndarray) -> np.ndarray:
        for stage in self.stages:
            data = await stage.process(data)
        return data
```

3.2 Metadata Extraction

Knowledge Graph Integration

```python
class MetadataExtractor:
    def __init__(self):
        self.kg_client = KnowledgeGraphClient(
            endpoint="http://kg-service:8080",
            auth=KGAuth()
        )
        self.linkml_validator = LinkMLValidator()

    async def extract_and_enrich(self, raw_metadata: dict) -> EnrichedMetadata:
        # 1. Extract from file headers
        file_metadata = self.extract_from_headers()

        # 2. Query knowledge graph
        kg_suggestions = await self.kg_client.query_similar(file_metadata)

        # 3. Apply LinkML schema
        validated = self.linkml_validator.validate(file_metadata)

        # 4. Mark provenance
        return EnrichedMetadata(
            user_provided=raw_metadata,
            extracted=file_metadata,
            ai_suggested=kg_suggestions,
            provenance=self.create_provenance()
        )
```

LinkML Schema Definition

```yaml
```

```yaml
# nwb_metadata_schema.yaml
id: https://example.org/nwb-metadata
name: nwb-metadata-schema
prefixes:
  nwb: https://www.nwb.org/

classes:
  Subject:
    attributes:
      subject_id:
        required: true
        range: string
      species:
        required: true
        range: Species
      age:
        range: string
        pattern: "^P\d+[YMD]$"  # ISO 8601 duration
      sex:
        range: Sex

  Device:
    attributes:
      name:
        required: true
      manufacturer:
        range: string
      model:
        range: string
```

3.3 Data Content Analysis

Signal Classification

python

```
class SignalClassifier:
    def __init__(self):
        self.models = {
            'spike_detector': SpikeDetectionModel(),
            'lfp_classifier': LFPClassifier(),
            'artifact_detector': ArtifactDetector(),
            'event_detector': EventDetector()
        }

    async def classify_signals(self, data: np.ndarray) -> SignalClassification:
        results = {}
        for name, model in self.models.items():
            results[name] = await model.classify(data)

        return SignalClassification(
            signal_types=results,
            confidence_scores=self.calculate_confidence(results),
            semantic_labels=self.generate_labels(results)
        )
```

3.4 Output and Reporting

Report Generation

```python
class ReportGenerator:
    def __init__(self):
        self.template_engine = Jinja2Engine()

    async def generate_reports(self, conversion_result: ConversionResult):
        reports = {
            'quality_report': self.generate_quality_report(conversion_result),
            'provenance_log': self.generate_provenance_log(conversion_result),
            'validation_report': self.generate_validation_report(conversion_result),
            'user_summary': self.generate_user_summary(conversion_result)
        }

        return reports
```

Quality Report Structure

```json
```

```
{
  "report_id": "uuid",
  "timestamp": "2024-01-01T00:00:00Z",
  "dataset_info": {
    "original_format": "string",
    "file_count": 0,
    "total_size_mb": 0.0
  },
  "conversion_metrics": {
    "duration_seconds": 0.0,
    "data_integrity": "preserved|modified|degraded",
    "metadata_completeness": 0.95
  },
  "validation_results": {
    "schema_compliance": true,
    "best_practices_score": 0.85,
    "warnings": [],
    "errors": []
  },
  "provenance": {
    "tool_versions": {},
    "user_modifications": [],
    "ai_suggestions_applied": []
  }
}
```

3.5 Evaluation and Refinement

Feedback Loop Implementation

```
python
```

```
class RefinementLoop:
    def __init__(self):
        self.max_iterations = 5
        self.improvement_threshold = 0.1

    async def refine(self, nwb_file: NWBFile, errors: List[ValidationError]):
        iteration = 0
        current_score = self.calculate_quality_score(nwb_file)

        while iteration < self.max_iterations:
            # Attempt automatic fixes
            auto_fixed = await self.apply_auto_fixes(nwb_file, errors)

            # Request human input for critical errors
            if self.has_critical_errors(errors):
                human_fixes = await self.request_human_input(errors)
                nwb_file = self.apply_human_fixes(nwb_file, human_fixes)

            # Re-validate
            new_errors = await self.validate(nwb_file)
            new_score = self.calculate_quality_score(nwb_file)

            # Check improvement
            if new_score - current_score < self.improvement_threshold:
                break

            errors = new_errors
            current_score = new_score
            iteration += 1

        return nwb_file, errors
```

---

4. Tools and Libraries Integration

4.1 NeuroConv Tool Integration

Configuration

```
python
```

```
class NeuroConvIntegration:
    def __init__(self):
        self.config = {
            "version": "0.4.0",
            "interfaces": {
                "blackrock": {
                    "enabled": true,
                    "options": {}
                },
                "intan": {
                    "enabled": true,
                    "options": {}
                }
            },
            "conversion_options": {
                "stub_test": false,
                "verbose": true,
                "compression": "gzip",
                "compression_opts": 4
            }
        }
```

Custom DataInterface Implementation

```python
class CustomDataInterface(BaseDataInterface):
    def __init__(self, file_path: str, **kwargs):
        super().__init__(file_path=file_path, **kwargs)

    def get_metadata(self) -> dict:
        # Extract metadata specific to custom format
        pass

    def get_metadata_schema(self) -> dict:
        # Return LinkML schema for validation
        pass

    def run_conversion(self, nwbfile: NWBFile, metadata: dict):
        # Implement conversion logic
        pass
```

4.2 Knowledge Graph Integration

Graph Database Schema

cypher

```cypher
// Neo4j schema for neuroscience metadata
CREATE CONSTRAINT subject_id ON (s:Subject) ASSERT s.id IS UNIQUE;
CREATE CONSTRAINT device_name ON (d:Device) ASSERT d.name IS UNIQUE;
CREATE CONSTRAINT experiment_id ON (e:Experiment) ASSERT e.id IS UNIQUE;

// Relationships
(s:Subject)-[:PARTICIPATED_IN]->(e:Experiment)
(e:Experiment)-[:USED_DEVICE]->(d:Device)
(e:Experiment)-[:HAS_RECORDING]->(r:Recording)
(r:Recording)-[:HAS_METADATA]->(m:Metadata)
```

Query Interface

python

```python
class KnowledgeGraphQuery:
    def __init__(self):
        self.driver = GraphDatabase.driver(
            "bolt://localhost:7687",
            auth=("neo4j", "password")
        )

    async def find_similar_experiments(self, metadata: dict) -> List[dict]:
        query = """
        MATCH (e:Experiment)-[:HAS_METADATA]->(m:Metadata)
        WHERE m.species = $species
        AND m.recording_type = $recording_type
        RETURN e, m
        LIMIT 10
        """

        with self.driver.session() as session:
            result = session.run(query, metadata)
            return [record.data() for record in result]
```

4.3 NWB Inspector Integration

Custom Validation Rules

python

```
class CustomValidationRules:
    def __init__(self):
        self.rules = [
            {
                "name": "electrode_table_completeness",
                "severity": "warning",
                "check": self.check_electrode_table
            },
            {
                "name": "timeseries_sampling_rate",
                "severity": "error",
                "check": self.check_sampling_rates
            }
        ]

    def check_electrode_table(self, nwbfile: NWBFile) -> ValidationResult:
        # Check if all required electrode properties are present
        pass

    def check_sampling_rates(self, nwbfile: NWBFile) -> ValidationResult:
        # Verify sampling rates are consistent
        pass
```

5. MCP Communication Protocol

5.1 Message Format

json

```json
{
  "jsonrpc": "2.0",
  "id": "unique-request-id",
  "method": "agent/action",
  "params": {
    "session_id": "session-uuid",
    "data": {},
    "metadata": {},
    "options": {}
  }
}
```

5.2 Inter-Agent Communication Flow

python

```
class MCPOrchestrator:
    def __init__(self):
        self.agents = {
            "conversation": ConversationAgentMCP(),
            "generation": GenerationAgentMCP(),
            "evaluation": EvaluationAgentMCP(),
            "tuie": TUIEAgentMCP()
        }
        self.message_bus = MessageBus()

    async def process_conversion(self, request: ConversionRequest):
        # 1. Conversation phase
        metadata = await self.agents["conversation"].gather_metadata(request)

        # 2. Generation phase
        nwb_file = await self.agents["generation"].convert(
            request.data_path,
            metadata
        )

        # 3. Evaluation phase
        validation = await self.agents["evaluation"].validate(nwb_file)

        # 4. TUIE monitoring (runs in parallel)
        metrics = await self.agents["tuie"].monitor_performance()

        # 5. Refinement if needed
        if validation.has_errors():
            nwb_file = await self.refine(nwb_file, validation.errors)

        return ConversionResult(
            nwb_file=nwb_file,
            validation=validation,
            metrics=metrics
        )
```

5.3 Error Handling

python

```
class MCPErrorHandler:
    def __init__(self):
        self.retry_policy = ExponentialBackoff(
            max_retries=3,
            base_delay=1.0
        )

    async def handle_error(self, error: MCPError):
        if error.is_retryable():
            return await self.retry_with_backoff(error.request)
        elif error.is_critical():
            await self.alert_admin(error)
            raise CriticalPipelineError(error)
        else:
            return self.create_error_response(error)
```

---

6. Deployment Architecture

6.1 Docker Compose Configuration

yaml

```
version: '3.8'

services:
  conversation-agent:
    build: ./agents/conversation
    environment:
      - MCP_SERVER_PORT=8001
      - POSTGRES_CONNECTION=${POSTGRES_CONNECTION}
    networks:
      - nwb-network

  generation-agent:
    build: ./agents/generation
    environment:
      - MCP_SERVER_PORT=8002
      - NEUROCONV_VERSION=0.4.0
    volumes:
      - ./data:/data
    networks:
      - nwb-network

  evaluation-agent:
    build: ./agents/evaluation
    environment:
      - MCP_SERVER_PORT=8003
      - NWB_INSPECTOR_CONFIG=/config/inspector.yaml
    networks:
      - nwb-network

  tuie-agent:
    build: ./agents/tuie
    environment:
      - MCP_SERVER_PORT=8004
      - MONITORING_ENABLED=true
    networks:
      - nwb-network

  orchestrator:
    build: ./orchestrator
    depends_on:
      - conversation-agent
      - generation-agent
      - evaluation-agent
      - tuie-agent
    ports:
      - "8000:8000"
    networks:
      - nwb-network

  knowledge-graph:
    image: neo4j:5.0
    environment:
      - NEO4J_AUTH=neo4j/password
    ports:
      - "7474:7474"
      - "7687:7687"
    volumes:
      - neo4j-data:/data
    networks:
      - nwb-network

  message-queue:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - nwb-network

networks:
  nwb-network:
    driver: bridge

volumes:
  neo4j-data:
  postgres-data:
```

6.2 Kubernetes Deployment

```yaml
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nwb-conversion-pipeline
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nwb-pipeline
  template:
    metadata:
      labels:
        app: nwb-pipeline
    spec:
      containers:
      - name: orchestrator
        image: nwb-pipeline/orchestrator:latest
        ports:
        - containerPort: 8000
        env:
        - name: MCP_AGENTS
          value: "conversation,generation,evaluation,tuie"
      - name: conversation-agent
        image: nwb-pipeline/conversation-agent:latest
        ports:
        - containerPort: 8001
      # Additional containers for other agents
```

---

7. Monitoring and Observability

7.1 Metrics Collection

```python
```

```python
class MetricsCollector:
    def __init__(self):
        self.prometheus_client = PrometheusClient()
        self.metrics = {
            'conversion_duration': Histogram('conversion_duration_seconds'),
            'validation_errors': Counter('validation_errors_total'),
            'metadata_completeness': Gauge('metadata_completeness_ratio'),
            'agent_response_time': Histogram('agent_response_seconds')
        }

    async def record_conversion(self, result: ConversionResult):
        self.metrics['conversion_duration'].observe(result.duration)
        self.metrics['validation_errors'].inc(len(result.errors))
        self.metrics['metadata_completeness'].set(result.metadata_score)
```

7.2 Logging Configuration

python

```
import structlog

logger = structlog.get_logger()

logging_config = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "json": {
            "()": structlog.stdlib.ProcessorFormatter,
            "processor": structlog.processors.JSONRenderer(),
        }
    },
    "handlers": {
        "default": {
            "class": "logging.StreamHandler",
            "formatter": "json",
        },
        "file": {
            "class": "logging.handlers.RotatingFileHandler",
            "filename": "nwb_pipeline.log",
            "maxBytes": 10485760,
            "backupCount": 5,
            "formatter": "json",
        }
    }
}
```

---

8. Security Considerations

8.1 Authentication & Authorization

python

```
class SecurityManager:
    def __init__(self):
        self.auth_provider = OAuth2Provider()
        self.rbac = RoleBasedAccessControl()

    async def authenticate(self, token: str) -> User:
        user = await self.auth_provider.verify_token(token)
        permissions = self.rbac.get_permissions(user.role)
        return AuthenticatedUser(user, permissions)

    def authorize(self, user: User, action: str, resource: str) -> bool:
        return self.rbac.check_permission(user, action, resource)
```

8.2 Data Encryption

python

```
class DataEncryption:
    def __init__(self):
        self.key_manager = KeyManager()

    def encrypt_at_rest(self, data: bytes) -> bytes:
        key = self.key_manager.get_encryption_key()
        return encrypt(data, key)

    def setup_tls(self):
        return {
            "cert_file": "/certs/server.crt",
            "key_file": "/certs/server.key",
            "ca_file": "/certs/ca.crt",
            "verify_mode": ssl.CERT_REQUIRED
        }
```

---

9. Testing Strategy

9.1 Unit Testing

python

```
import pytest

class TestConversionAgent:
    @pytest.fixture
    def agent(self):
        return GenerationAgentMCP()

    async def test_conversion_success(self, agent, sample_data):
        result = await agent.convert_dataset(sample_data)
        assert result.status == "success"
        assert Path(result.nwb_file_path).exists()

    async def test_metadata_injection(self, agent, metadata):
        nwb_file = await agent.inject_metadata(metadata)
        assert nwb_file.subject.subject_id == metadata["subject_id"]
```

9.2 Integration Testing

```python
class TestPipelineIntegration:
    async def test_end_to_end_conversion(self):
        # Setup test data
        test_data = create_test_dataset()

        # Run full pipeline
        orchestrator = MCPOrchestrator()
        result = await orchestrator.process_conversion(test_data)

        # Validate results
        assert result.validation.is_valid()
        assert result.metrics.performance_score > 0.8
```

---

10. Performance Optimization

10.1 Caching Strategy

```python
```

```
class CacheManager:
    def __init__(self):
        self.redis_client = Redis()
        self.cache_ttl = 3600  # 1 hour

    async def get_or_compute(self, key: str, compute_func):
        cached = await self.redis_client.get(key)
        if cached:
            return pickle.loads(cached)

        result = await compute_func()
        await self.redis_client.setex(
            key,
            self.cache_ttl,
            pickle.dumps(result)
        )
        return result
```

10.2 Parallel Processing

python

```
class ParallelProcessor:
    def __init__(self):
        self.executor = ProcessPoolExecutor(max_workers=4)

    async def process_batch(self, files: List[str]):
        tasks = []
        for file in files:
            task = self.executor.submit(self.process_file, file)
            tasks.append(task)

        results = await asyncio.gather(*tasks)
        return results
```

---

11. Maintenance and Updates

11.1 Version Management

python

```
class VersionManager:
    def __init__(self):
        self.versions = {
            "pipeline": "1.0.0",
            "neuroconv": "0.4.0",
            "nwb_schema": "2.6.0",
            "mcp_protocol": "1.0"
        }

    def check_compatibility(self) -> bool:
        # Verify all components are compatible
        pass

    def upgrade_strategy(self, target_version: str) -> UpgradePlan:
        # Generate upgrade plan
        pass
```

11.2 Backup and Recovery

python

```
class BackupManager:
    def __init__(self):
        self.backup_storage = S3Storage()

    async def backup_conversion(self, conversion_id: str):
        data = {
            "timestamp": datetime.now(),
            "input_data": self.get_input_data(conversion_id),
            "output_data": self.get_output_data(conversion_id),
            "metadata": self.get_metadata(conversion_id),
            "logs": self.get_logs(conversion_id)
        }

        await self.backup_storage.upload(
            f"backups/{conversion_id}.tar.gz",
            self.compress(data)
        )
```

---

12. User Interface Specifications

12.1 Web UI Components

typescript

```
// React components for frontend
interface ConversionDashboard {
    sessionManager: SessionManager;
    metadataEditor: MetadataEditor;
    progressTracker: ProgressTracker;
    validationViewer: ValidationViewer;
    reportDownloader: ReportDownloader;
}

interface MetadataEditor {
    fields: MetadataField[];
    validation: ValidationRules;
    aiSuggestions: Suggestion[];
    onSave: (metadata: Metadata) => void;
}
```

12.2 CLI Interface

```bash
# Command-line interface
nwb-convert \
    --input /path/to/data \
    --output /path/to/output \
    --metadata metadata.json \
    --config config.yaml \
    --validate \
    --verbose

# Agent-specific commands
nwb-agent conversation --start --session-id abc123
nwb-agent generation --convert --input data.dat
nwb-agent evaluation --validate output
```