```
def setup_api_settings_ui(self, parent_frame):
"""
Create UI for API settings configuration
"""
# Create notebook for API settings
api_notebook = ttk.Notebook(parent_frame)
api_notebook.pack(fill="both", expand=True, padx=5, pady=5)
```

```
# OMDB API Keys tab
omdb_tab = ttk.Frame(api_notebook)
api_notebook.add(omdb_tab, text="OMDB API")

ttk.Label(omdb_tab, text="OMDB API Keys (Up to 5)").pack(anchor="w", padx=5, pady=5)

# Frame for OMDB keys with scrollbar
omdb_frame = ttk.Frame(omdb_tab)
omdb_frame.pack(fill="both", expand=True, padx=5, pady=5)

omdb_canvas = tk.Canvas(omdb_frame)
scrollbar = ttk.Scrollbar(omdb_frame, orient="vertical", command=omdb_canvas.yview)
scrollable_frame = ttk.Frame(omdb_canvas)

scrollable_frame.bind(
    "<Configure>",
    lambda e: omdb_canvas.configure(scrollregion=omdb_canvas.bbox("all"))
)

omdb_canvas.create_window((0, 0), window=scrollable_frame, anchor="nw")
omdb_canvas.configure(yscrollcommand=scrollbar.set)

omdb_canvas.pack(side="left", fill="both", expand=True)
scrollbar.pack(side="right", fill="y")

# OMDB API key entry fields
self.omdb_keys = []
for i in range(5):  # Up to 5 OMDB keys
    key_frame = ttk.Frame(scrollable_frame)
    key_frame.pack(fill="x", padx=5, pady=2)

    ttk.Label(key_frame, text=f"API Key {i+1}:").pack(side="left")
    key_var = tk.StringVar()
    key_entry = ttk.Entry(key_frame, textvariable=key_var, width=40)
    key_entry.pack(side="left", padx=5)

    status_var = tk.StringVar(value="Not used")
    ttk.Label(key_frame, textvariable=status_var).pack(side="left", padx=5)

    self.omdb_keys.append((key_var, status_var))

# Load existing OMDB keys
self.load_api_keys("omdb")

# OMDB Settings
ttk.Separator(omdb_tab).pack(fill="x", padx=5, pady=10)
```

```
ttk.Label(omdb_tab, text="OMDB API Settings").pack(anchor="w", padx=5, pady=5)

settings_frame = ttk.Frame(omdb_tab)
settings_frame.pack(fill="x", padx=5, pady=5)

ttk.Label(settings_frame, text="Auto-rotate keys:").pack(side="left")
auto_rotate = tk.BooleanVar(value=True)
ttk.Checkbutton(settings_frame, variable=auto_rotate).pack(side="left", padx=5)

ttk.Label(settings_frame, text="Current Status:").pack(side="left", padx=10)
status_label = ttk.Label(settings_frame, text="Active", foreground="green")
status_label.pack(side="left")

# Sonarr tab
sonarr_tab = ttk.Frame(api_notebook)
api_notebook.add(sonarr_tab, text="Sonarr")
self.setup_arr_interface(sonarr_tab, "sonarr", 10)

# Radarr tab
radarr_tab = ttk.Frame(api_notebook)
api_notebook.add(radarr_tab, text="Radarr")
self.setup_arr_interface(radarr_tab, "radarr", 10)

# API Status tab
status_tab = ttk.Frame(api_notebook)
api_notebook.add(status_tab, text="API Status")
self.setup_api_status_ui(status_tab)
```

def setup_arr_interface(self, parent, service_type, max_instances):
"""
Create UI for Sonarr/Radarr instance management
"""
service_name = service_type.capitalize()
ttk.Label(parent, text=f"{service_name} Instances (Up to {max_instances})").pack(anchor="w", padx=5, pady=5)

```
    # Add instance button
    add_button = ttk.Button(
        parent,
        text=f"Add {service_name} Instance",
        command=lambda: self.add_arr_instance(service_type)
    )
    add_button.pack(anchor="w", padx=5, pady=5)

    # Frame for instances with scrollbar
    instances_frame = ttk.Frame(parent)
    instances_frame.pack(fill="both", expand=True, padx=5, pady=5)

    canvas = tk.Canvas(instances_frame)
    scrollbar = ttk.Scrollbar(instances_frame, orient="vertical", command=canvas.yview)
    scrollable_frame = ttk.Frame(canvas)

    scrollable_frame.bind(
        "<Configure>",
        lambda e: canvas.configure(scrollregion=canvas.bbox("all"))
    )

    canvas.create_window((0, 0), window=scrollable_frame, anchor="nw")
    canvas.configure(yscrollcommand=scrollbar.set)

    canvas.pack(side="left", fill="both", expand=True)
    scrollbar.pack(side="right", fill="y")

    # Set attribute to store the scrollable frame
    setattr(self, f"{service_type}_frame", scrollable_frame)

    # Load existing instances
    self.load_arr_instances(service_type)
```

def add_arr_instance(self, service_type):
"""
Add a new Sonarr/Radarr instance to the UI
"""
# Get the scrollable frame
scrollable_frame = getattr(self, f"{service_type}_frame")

```
# Count existing instances
existing_count = len([w for w in scrollable_frame.winfo_children() if isinstance(w,
ttk.LabelFrame)])

# Maximum check
if existing_count >= 10:
    tk.messagebox.showerror("Maximum Reached", f"You can only have up to 10
{service_type.capitalize()} instances")
    return

# Create instance frame
instance_frame = ttk.LabelFrame(scrollable_frame, text=f"Instance {existing_count + 1}")
instance_frame.pack(fill="x", padx=5, pady=5)

# Name
name_frame = ttk.Frame(instance_frame)
name_frame.pack(fill="x", padx=5, pady=2)
ttk.Label(name_frame, text="Name:").pack(side="left")
name_var = tk.StringVar(value=f"{service_type.capitalize()} {existing_count + 1}")
ttk.Entry(name_frame, textvariable=name_var).pack(side="left", padx=5, fill="x",
expand=True)

# URL
url_frame = ttk.Frame(instance_frame)
url_frame.pack(fill="x", padx=5, pady=2)
ttk.Label(url_frame, text="URL:").pack(side="left")
url_var = tk.StringVar()
ttk.Entry(url_frame, textvariable=url_var).pack(side="left", padx=5, fill="x", expand=True)

# API Key
key_frame = ttk.Frame(instance_frame)
key_frame.pack(fill="x", padx=5, pady=2)
ttk.Label(key_frame, text="API Key:").pack(side="left")
key_var = tk.StringVar()
ttk.Entry(key_frame, textvariable=key_var).pack(side="left", padx=5, fill="x", expand=True)

# Buttons
btn_frame = ttk.Frame(instance_frame)
btn_frame.pack(fill="x", padx=5, pady=5)

ttk.Button(
    btn_frame,
    text="Test Connection",
    command=lambda: self.test_arr_connection(name_var.get(), url_var.get(), key_var.get())
).pack(side="left", padx=5)
```

```
    ttk.Button(
        btn_frame,
        text="Remove",
        command=lambda: self.remove_arr_instance(instance_frame, service_type)
    ).pack(side="right", padx=5)
```

def test_arr_connection(self, name, url, api_key):
"""
Test connection to a Sonarr/Radarr instance
"""
try:
# Simple test request
headers = {
"X-Api-Key": api_key,
"Content-Type": "application/json"
}
response = requests.get(f"{url.rstrip('/')}/api/system/status", headers=headers, timeout=10)

```
        if response.status_code == 200:
            tk.messagebox.showinfo("Connection Test", f"Successfully connected to {name}!")
        else:
            tk.messagebox.showerror("Connection Failed", f"Error connecting to {name}:
    {response.status_code}")
    except Exception as e:
        tk.messagebox.showerror("Connection Failed", f"Error connecting to {name}: {str(e)}")
```

def setup_api_status_ui(self, parent):
"""
Create UI for API status monitoring
"""
# Create header
ttk.Label(parent, text="API Status Dashboard", font=("TkDefaultFont", 12, "bold")).pack(anchor="w", padx=5, pady=5)

```
    # Create treeview for API status
    columns = ("service", "name", "status", "usage", "last_used", "errors")
    tree = ttk.Treeview(parent, columns=columns, show="headings")

    # Define headings
    tree.heading("service", text="Service")
    tree.heading("name", text="Name")
    tree.heading("status", text="Status")
    tree.heading("usage", text="Usage")
    tree.heading("last_used", text="Last Used")
    tree.heading("errors", text="Errors")

    # Define columns
    tree.column("service", width=100)
    tree.column("name", width=150)
    tree.column("status", width=100)
    tree.column("usage", width=100)
    tree.column("last_used", width=150)
    tree.column("errors", width=100)

    # Add scrollbar
    scrollbar = ttk.Scrollbar(parent, orient="vertical", command=tree.yview)
    tree.configure(yscrollcommand=scrollbar.set)

    # Pack elements
    tree.pack(side="left", fill="both", expand=True, padx=5, pady=5)
    scrollbar.pack(side="right", fill="y", pady=5)

    # Refresh button
    ttk.Button(
        parent,
        text="Refresh Status",
        command=lambda: self.refresh_api_status(tree)
    ).pack(anchor="center", pady=10)

    # Store reference to the tree
    self.api_status_tree = tree

    # Initial refresh
    self.refresh_api_status(tree)
```

def refresh_api_status(self, tree):
"""
Refresh the API status display
"""

# Clear existing items
for item in tree.get_children():
tree.delete(item)

```
# Get all API keys from database
cursor = self.db_conn.cursor()
cursor.execute(
    "SELECT service, instance_name, is_active, daily_uses, last_used, error_count "
    "FROM api_keys ORDER BY service, instance_name"
)

for service, name, is_active, usage, last_used, errors in cursor.fetchall():
    # Format data
    status = "Active" if is_active else "Inactive"

    # Format last used time
    if last_used:
        last_used_str = datetime.fromtimestamp(last_used).strftime("%Y-%m-%d %H:%M:%S")
    else:
        last_used_str = "Never"

    # Format usage for OMDB
    if service == "omdb":
        usage_str = f"{usage}/1000"
    else:
        usage_str = str(usage)

    # Insert into tree
    tree.insert("", "end", values=(service.upper(), name, status, usage_str, last_used_str,
errors))

# Add OMDB cooldown status if applicable
cursor.execute("SELECT value FROM settings WHERE key = 'omdb_cooldown_end'")
result = cursor.fetchone()

if result:
    cooldown_end = float(result[0])
    now = datetime.now().timestamp()

    if cooldown_end > now:
        # Still in cooldown
        end_time = datetime.fromtimestamp(cooldown_end).strftime("%Y-%m-%d %H:%M:%S")
        tree.insert("", "end", values=("OMDB", "API COOLDOWN", "Waiting", "", f"Until
{end_time}", "Rate Limited"))
```

API Management System

The application will include a sophisticated API management system to handle multiple API keys and service instances, with the following features:

OMDB API Key Management

- Support for up to 5 OMDB API keys
- Automatic key rotation when 1000 requests/24h limit is reached
- Usage tracking per key with timestamps
- Error detection and key deactivation on API errors
- 24-hour cooldown timer when all keys are exhausted
- Automatic resume after cooldown period

Sonarr/Radarr Instance Management

- Support for up to 10 Sonarr instances
- Support for up to 10 Radarr instances
- Individual configuration for each instance (URL, API key)
- Connection testing for each instance
- Fallback mechanism if an instance is unavailable
- Priority setting for instance usage order

API Request Queue

- Prioritized queue for API requests
- Rate limiting to prevent API overload
- Request batching where possible
- Error handling with exponential backoff

Implementation Details

API Key Rotation Algorithm

python

```
def get_available_omdb_api_key(self):
    """
    Get the next available OMDB API key, respecting rate limits.
    Returns the API key or None if all keys are exhausted.
    """
    current_date = datetime.now().strftime("%Y-%m-%d")

    # Get all active OMDB API keys
    cursor = self.db_conn.cursor()
    cursor.execute(
        "SELECT id, api_key, daily_uses, last_reset_date, error_count FROM api_keys "
        "WHERE service = 'omdb' AND is_active = 1 ORDER BY daily_uses ASC"
    )
    keys = cursor.fetchall()

    if not keys:
        return None

    for key_id, api_key, daily_uses, last_reset_date, error_count in keys:
        # Reset counter if date changed
        if last_reset_date != current_date:
            cursor.execute(
                "UPDATE api_keys SET daily_uses = 0, last_reset_date = ? WHERE id = ?",
                (current_date, key_id)
            )
            daily_uses = 0

        # Skip this key if it has errors or reached limit
        if error_count >= 3:
            continue

        if daily_uses < 1000:  # OMDB daily limit
            # Update usage counter
            cursor.execute(
                "UPDATE api_keys SET daily_uses = daily_uses + 1, last_used = ? WHERE id = ?",
                (datetime.now().timestamp(), key_id)
            )
            self.db_conn.commit()
            return api_key

    # If we get here, all keys are exhausted
    self.logger.warning("All OMDB API keys are exhausted or have errors")
    return None

def handle_api_error(self, service, api_key, error_message):
    """
```

```
    Handle API error by incrementing error count and potentially deactivating key
    """
    cursor = self.db_conn.cursor()

    # Find the key
    cursor.execute(
        "SELECT id, error_count FROM api_keys WHERE service = ? AND api_key = ?",
        (service, api_key)
    )
    result = cursor.fetchone()

    if result:
        key_id, error_count = result
        new_error_count = error_count + 1

        # Update error information
        cursor.execute(
            "UPDATE api_keys SET error_count = ?, last_error = ? WHERE id = ?",
            (new_error_count, error_message, key_id)
        )

        # If all OMDB keys have errors, schedule a cooldown
        if service == 'omdb':
            cursor.execute(
                "SELECT COUNT(*) FROM api_keys WHERE service = 'omdb' AND error_count < 3 AND i
            )
            active_keys = cursor.fetchone()[0]

            if active_keys == 0:
                # Schedule 24h cooldown
                self.schedule_api_cooldown('omdb', 24 * 60 * 60)  # 24 hours in seconds

        self.db_conn.commit()

def schedule_api_cooldown(self, service, cooldown_seconds):
    """
    Schedule an API cooldown period
    """
    cooldown_end = datetime.now() + timedelta(seconds=cooldown_seconds)

    # Store cooldown info in settings
    cursor = self.db_conn.cursor()
    cursor.execute(
        "INSERT OR REPLACE INTO settings (key, value) VALUES (?, ?)",
        (f"{service}_cooldown_end", cooldown_end.timestamp())
    )
    self.db_conn.commit()
```

```
        self.logger.warning(f"{service.upper()} API on cooldown until {cooldown_end}")

        # Schedule reset task
        threading.Timer(cooldown_seconds, self.reset_api_errors, args=[service]).start()

    def reset_api_errors(self, service):
        """
        Reset API errors after cooldown period
        """
        cursor = self.db_conn.cursor()
        cursor.execute(
            "UPDATE api_keys SET error_count = 0 WHERE service = ?",
            (service,)
        )
        self.db_conn.commit()
        self.logger.info(f"{service.upper()} API cooldown period ended, errors reset")
```

Sonarr/Radarr Instance Selection

python

```python
def get_metadata_from_instances(self, tt_id, content_type):
    """
    Try to get metadata from configured instances
    """
    if content_type.lower() == 'movie':
        service = 'radarr'
    else:  # TV show
        service = 'sonarr'

    # Get instances sorted by priority
    cursor = self.db_conn.cursor()
    cursor.execute(
        "SELECT id, instance_name, url, api_key FROM api_keys "
        "WHERE service = ? AND is_active = 1 ORDER BY id ASC",
        (service,)
    )
    instances = cursor.fetchall()

    if not instances:
        self.logger.warning(f"No {service} instances configured")
        return None

    # Try each instance until we get a result
    for instance_id, name, url, api_key in instances:
        try:
            result = self.query_arr_instance(url, api_key, tt_id)
            if result:
                return result
        except Exception as e:
            self.logger.error(f"Error querying {service} instance {name}: {str(e)}")
            continue

    return None
```**API Keys Table**

```
CREATE TABLE api_keys (
id INTEGER PRIMARY KEY AUTOINCREMENT,
service TEXT,  -- 'omdb', 'sonarr', 'radarr'
instance_name TEXT,
url TEXT,
api_key TEXT,
last_used TIMESTAMP,
daily_uses INTEGER DEFAULT 0,
```

```
last_reset_date TEXT,
last_error TEXT,
error_count INTEGER DEFAULT 0,
is_active BOOLEAN DEFAULT 1
);
```

   - OMDB API key rotation and rate limiting
   - Multiple Sonarr/Radarr instance management
   - API error handling and cooldown logic
   - Usage statistics tracking# Media Processing Application
Technical Specification Document

Project Overview

This document specifies the requirements and implementation details for a distributed media processing application that monitors directories for media files, processes them according to specific rules, and maintains a queue system with statistics. The application will be built using Python with a Tkinter GUI interface and packaged as a standalone executable.

Core Functionality

1. **File Monitoring**
   - Monitor specified directories for new media files
   - Identify file types and apply appropriate processing rules
   - Support distributed monitoring across multiple machines

2. **Media Analysis**
   - Analyze media files to identify video codecs, audio tracks, and subtitles
   - Detect Dolby Vision HDR content
   - Identify audio track languages and channel configurations
   - Check subtitle attributes

3. **Metadata Integration**
   - Interface with Sonarr/Radarr to fetch metadata using tt{idnumber}
   - Connect to OMDB API to determine native audio language
   - Use metadata to make processing decisions

4. **Audio Processing**
   - Select audio tracks based on language preferences (native language + eng,dut/nld,tur,und)
   - Remove commentary tracks
   - Convert audio to Opus with bitrate decisions based on channel configuration:
     - 1.0 → 1.0 Opus 32kbps
     - 2.0 → 2.0 Opus 64kbps
     - 2.1, 3.0, 4.0 → 2.0 Opus 64kbps
     - 5.1 (128-384kbps) → 5.1 Opus 128kbps
     - 5.1 (384-640kbps) → 5.1 Opus 256kbps
     - 5.1 (>640kbps) → 5.1 Opus 320kbps
     - 7.1/9.1 (128-384kbps) → 5.1 Opus 128kbps
     - 7.1/9.1 (384-640kbps) → 5.1 Opus 256kbps
     - 7.1/9.1 (>640kbps) → 5.1 Opus 320kbps

5. **Subtitle Management**
   - Keep subtitles in eng, dut/nld, tur, und languages
   - Remove SDH and commentary subtitles
   - Reorder subtitle tracks

6. **Container Management**
   - Run mkvpropedit on the new container
   - Reorder streams (video, audio by language priority and channel count, subtitles by language priority)

7. **Distributed Processing**
   - Inter-instance communication between applications running on different machines
   - Status sharing and coordination
   - Work distribution

8. **Queue Management**
   - Queuing system for pending jobs
   - Currently processing view
   - Completed jobs history with statistics
   - Failed jobs tracking
   - Configurable parallel processing limits per instance

9. **Customizable Commands**
   - User-definable command templates for different content types:
     - Dolby Vision HDR content processing
     - Normal content processing
     - Downscaling content (for files with downscale tag)

10. **User Interface**
    - Start/stop/pause controls
    - Queue management interface
    - Processing status and progress
    - Configuration interface
    - Log viewer
    - Statistics display

11. **Logging & Statistics**
    - Detailed logging of all operations
    - HTML log file generation
    - Processing statistics tracking and persistence
    - Job history maintenance
    - Restart-persistent application state

Technical Implementation

Programming Language & Framework
- Python 3.8+ with Tkinter GUI

Required Python Libraries
- **Built-in Libraries**
  - `tkinter` - GUI framework
  - `threading` - Background processing
  - `queue` - Thread-safe queue implementation
  - `sqlite3` - Local database for settings and statistics
  - `json` - Configuration and state serialization
  - `logging` - Application logging
  - `os`, `sys`, `shutil` - File system operations
  - `socket` - Basic networking
  - `subprocess` - Executing external commands
  - `string` - String templating for commands

- **External Libraries (pip installable)**
  - `watchdog` - File system monitoring
  - `pymediainfo` - MediaInfo wrapper for media file analysis
  - `ffmpeg-python` - FFmpeg wrapper for audio conversion
  - `requests` - HTTP client for API calls
  - `websockets` - Inter-instance communication (alternative to raw sockets)
  - `pyinstaller` - Executable creation

External Dependencies
- FFmpeg - Media processing toolkit
- MediaInfo - Media file analysis
- qsvenc - Video encoding
- mkvpropedit - MKV container manipulation

Application Architecture

1. **Main Application (main.py)**
   - Entry point
   - UI initialization
   - Background services startup

2. **User Interface (ui.py)**
   - Main window setup
   - Tabbed interface (Queue, Statistics, Logs, Settings)
   - Control panel
   - Status updates

3. **Configuration Manager (config.py)**
   - Settings loading/saving
   - Command templates management
   - UI preferences

4. **File Monitor (monitor.py)**

```
   - Directory watching
   - New file detection
   - File type identification

5. **Media Analyzer (analyzer.py)**
   - Media file inspection
   - Stream identification
   - Language detection
   - API integration (Sonarr/Radarr, OMDB)

6. **Processor (processor.py)**
   - Command execution
   - Audio conversion
   - Container manipulation
   - Progress tracking

7. **Queue Manager (queue_manager.py)**
   - Job scheduling
   - Parallel execution control
   - State persistence

8. **Network Communication (network.py)**
   - Inter-instance messaging
   - Status broadcasting
   - Work distribution

9. **Statistics Manager (statistics.py)**
   - Job history tracking
   - Performance metrics
   - Data visualization

10. **Logger (logger.py)**
    - Console logging
    - File logging
    - HTML log generation

#### Database Schema

**Settings Table**
```

```
CREATE TABLE settings (
key TEXT PRIMARY KEY,
value TEXT
);
```

Jobs Table

```
CREATE TABLE jobs (
id INTEGER PRIMARY KEY AUTOINCREMENT,
filename TEXT,
status TEXT,  -- 'queued', 'processing', 'completed', 'failed'
start_time TIMESTAMP,
end_time TIMESTAMP,
original_size INTEGER,
processed_size INTEGER,
command_used TEXT,
error_message TEXT,
instance_id TEXT
);
```

Statistics Table

```
CREATE TABLE statistics (
date TEXT,
jobs_completed INTEGER,
jobs_failed INTEGER,
total_size_processed INTEGER,
total_size_saved INTEGER,
average_processing_time REAL
);
```

User Interface Design

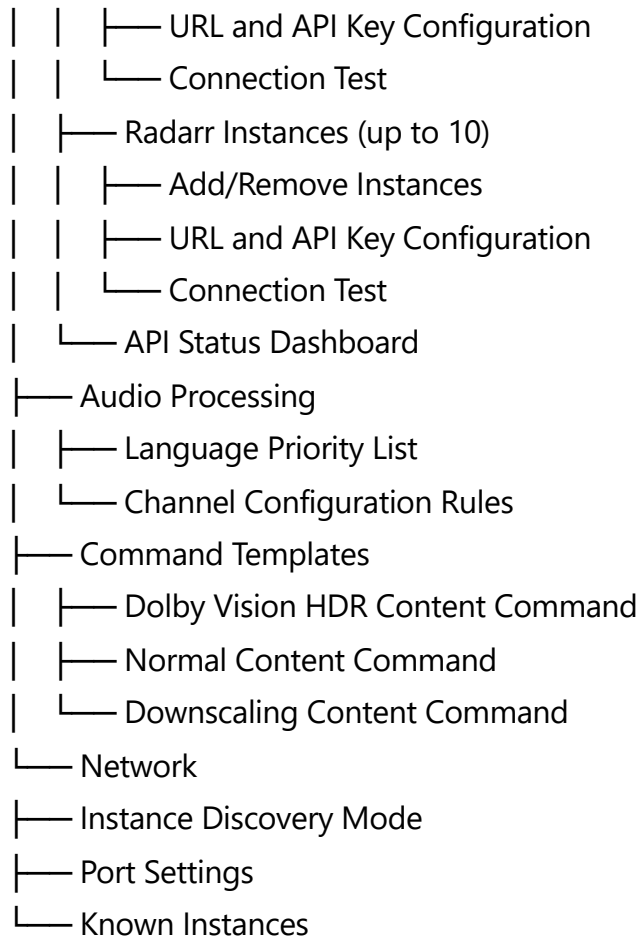
Main Window Layout

```
Main Window
├── Menu Bar (File, Settings, Help)
├── Control Panel (Start/Stop/Pause buttons)
├── Instance Mode Indicator (Master/Slave/Standalone)
├── Tab View
│   ├── Queue Tab
│   │   ├── Currently Processing Section
```

```
│   │   ├── Pending Jobs Section
│   │   └── Completed Jobs Section
│   ├── Statistics Tab
│   │   ├── Processing History
│   │   └── File Statistics Charts
│   ├── Network Tab (visible in Master mode)
│   │   ├── Connected Slaves List
│   │   ├── Instance Status Overview
│   │   └── Work Distribution Stats
│   ├── Logs Tab
│   │   └── Real-time Log Viewer
│   └── Settings Tab
│       ├── Folder Monitoring Settings
│       ├── Processing Rules Settings
│       ├── Network Settings
│       ├── Command Configuration
│       │   ├── Dolby Vision HDR Commands
│       │   ├── Standard Content Commands
│       │   └── Downscaling Commands
│       └── Output Settings
└── Status Bar (Processing status, instance info, connection status)
```

Settings Tab Structure

```
Settings Tab
├── General
│   ├── Instance Name
│   ├── Max Parallel Jobs
│   └── Default Output Directory
├── Monitoring
│   ├── Add/Remove Watched Directories
│   └── File Pattern Filters
├── API Integration
│   ├── OMDB API Keys Management (up to 5 keys)
│   │   ├── API Key Entry Fields
│   │   ├── Key Rotation Settings
│   │   └── Usage Statistics
│   ├── Sonarr Instances (up to 10)
│   │   ├── Add/Remove Instances
```


URL and API Key Configuration
Connection Test
Radarr Instances (up to 10)
Add/Remove Instances
URL and API Key Configuration
Connection Test
API Status Dashboard
Audio Processing
Language Priority List
Channel Configuration Rules
Command Templates
Dolby Vision HDR Content Command
Normal Content Command
Downscaling Content Command
Network
Instance Discovery Mode
Port Settings
Known Instances

```
#### Inter-Instance Communication Protocol

**Network Architecture**
- **Master Mode**: Central coordinator that distributes work and tracks status
- **Slave Mode**: Connects to a master instance for job coordination

**Message Types**
- `HELLO` - Instance discovery and capabilities announcement
- `STATUS` - Instance status update
- `JOB_REQUEST` - Request for job information
- `JOB_STATUS` - Job status update
- `JOB_COMPLETE` - Job completion notification with stats
- `SYSTEM_STATUS` - Overall system status report
- `REGISTER_SLAVE` - Slave instance registering with master
- `MASTER_COMMAND` - Command from master to slave
- `SLAVE_RESPONSE` - Response from slave to master

**Message Format**
```json
{
  "type": "MESSAGE_TYPE",
  "instance_id": "unique-instance-id",
  "instance_role": "master|slave",
  "timestamp": 1620000000,
  "data": {
    // Message-specific payload
  }
}
```
```

Master-Slave Connection Flow

1. Slave starts up and checks configuration
2. Slave attempts to connect to configured master IP/port
3. Slave sends `REGISTER_SLAVE` message with capabilities
4. Master acknowledges and adds slave to instance pool
5. Regular heartbeat messages maintain connection
6. Master coordinates job distribution across connected slaves

Implementation Plan

1. **Phase 1: Core Application Framework**
   - Set up basic Tkinter application structure
   - Implement settings management
   - Create basic UI layout

2. **Phase 2: File Monitoring & Analysis**
   - Implement directory watching
   - Build media file analysis module
   - Create file type detection logic

3. **Phase 3: Processing Engine**
   - Implement command template system
   - Build audio conversion logic
   - Create subtitle handling

4. **Phase 4: Queue Management**
   - Build job queuing system
   - Implement parallel processing
   - Create job status tracking

5. **Phase 5: Network Communication**
   - Implement instance discovery
   - Build message passing system
   - Create work distribution logic

6. **Phase 6: Statistics & Logging**
   - Implement detailed logging
   - Build statistics collection
   - Create HTML log generator

7. **Phase 7: UI Finalization**
   - Refine user interface
   - Add data visualization
   - Implement all configuration options

8. **Phase 8: Packaging & Testing**
   - Package application as executable
   - Perform comprehensive testing
   - Fix identified issues

Executable Creation

The application will be packaged using PyInstaller:

```bash
# Basic package
pyinstaller --onefile --windowed --icon=app_icon.ico --name=MediaProcessor main.py

# With additional data files
pyinstaller --onefile --windowed --icon=app_icon.ico --add-data "assets/*:assets" --
name=MediaProcessor main.py
```

System Requirements

- Python 3.8+ (development)
- Windows 7/10/11 (target deployment)
- FFmpeg, MediaInfo, qsvenc, and mkvpropedit installed
- Network connectivity for distributed mode
- Sufficient storage for media processing

Additional Notes

1. **Error Handling**
   - All external tool calls should have robust error handling
   - Failed jobs should be logged with detailed error information
   - Application should gracefully handle unexpected shutdowns
   - Network disconnections should be detected and recovered from
2. **Performance Considerations**
   - File analysis should be done efficiently to minimize startup time
   - Background processing should be properly threaded to keep UI responsive
   - Database operations should be batched where possible
   - Network communication should be optimized to reduce overhead
3. **Security**
   - API keys should be stored securely
   - Network communication should validate source instances
   - File paths should be sanitized before use in commands
   - Master-slave connections should implement basic authentication
4. **Extensibility**
   - Command template system allows for future expansion
   - Plugin architecture could be considered for future versions
   - Configuration system should be designed for easy extension

5. **Master-Slave Architecture**
    - Master instance coordinates work distribution
    - Slave instances report capabilities and status to master
    - Job allocation considers slave capabilities and current load
    - System handles instances joining and leaving dynamically
    - Master failure should be detected by slaves

This specification document provides a comprehensive overview of the Media Processing Application requirements, architecture, and implementation plan. It serves as a guide for development and can be expanded upon as needed during the implementation process.