Video to MP4 — Software Design Document (SDD)

- Project: `video_to_mp4`
- Version: `0.1.0`
- Tech Stack: Python 3.11, Reflex `0.8.24.post1`, Tailwind, `ffmpeg-python`, system `ffmpeg`
- Repo Root: `https://github.com/milochen0418/video_to_mp4/`

—

1. Overview

1.1 Purpose

This project is a lightweight web tool that converts uploaded videos (AVI/MOV/MKV/WMV/MP4/WEBM) into MP4. The user flow is intentionally simple:
1. Select/drag video files
2. Confirm conversion settings (resolution + quality)
3. Background conversion runs
4. Download MP4 result

1.2 Key Features

- Drag-and-drop multi-file upload
- Confirmation dialog before starting conversion
- Per-file job queue with statuses: Queued → Processing → Complete / Error
- Progress tracking driven by FFmpeg `-progress pipe:1`
- Download link for the converted MP4 via Reflex upload serving

1.3 Non-goals (Current)

- Persistent storage across server restarts (jobs are in memory)
- Authentication / multi-user isolation
- Cloud storage (S3/GCS) integration
- Distributed workers / job broker (Celery/Redis)

—

2. Repository Structure

```
video_to_mp4-main/
├── video_to_mp4/
│   ├── video_to_mp4.py              # App entry page (route "/")
│   ├── states/
│   │   └── app_state.py             # Core state + FFmpeg conversion logic
│   └── components/
│       ├── upload_zone.py           # Upload UI + confirm dialog + settings UI
│       ├── job_list.py              # Job table + download/retry/delete actions
│       ├── sidebar.py               # UI sidebar (not used on index page)
│       └── capacity_indicator.py    # Planned/unused (state fields not present)
├── rxconfig.py                      # Reflex config (Tailwind plugin)
├── pyproject.toml                   # Poetry deps
├── requirements.txt                 # Minimal runtime deps
├── apt-packages.txt                 # Includes ffmpeg
├── README.md                        # Screenshots + run instructions
└── plan.md                          # Implementation plan checklist
```

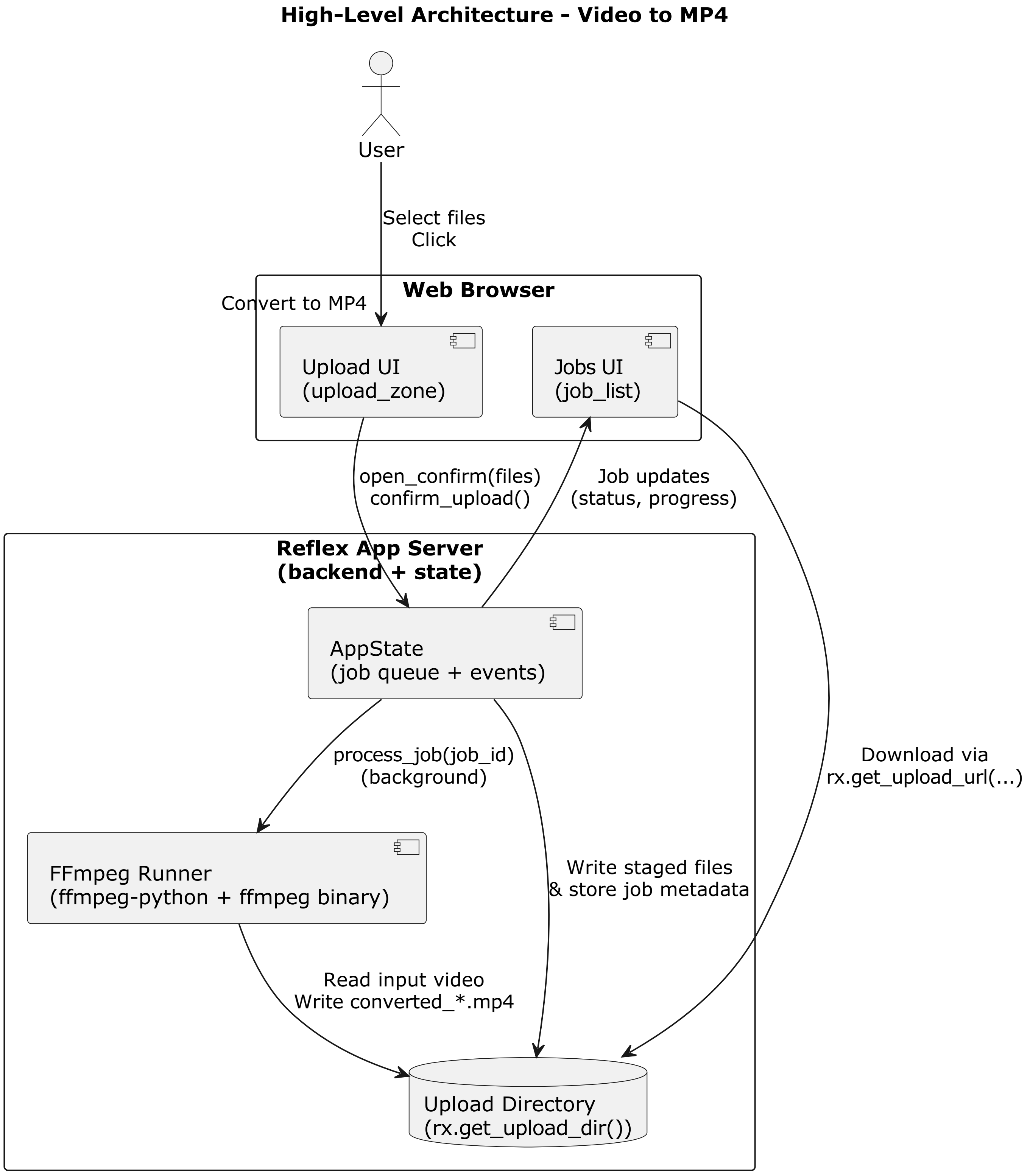
Notes:

- The UI currently renders `upload_zone()` + `job_list()` on the main page.
- `capacity_indicator.py` references state fields (like `MAX_CAPACITY_GB`) that do **not** exist in `AppState`; treat it as a planned feature or stale component.

—

3. System Architecture

3.1 High-Level Architecture (Browser ↔ Reflex ↔ FFmpeg ↔ File Storage)



3.2 Runtime Boundaries

- **Frontend**: Reflex-generated UI rendered in the browser.

- **Backend**: Reflex state machine (`AppState`) executes Python code and handles events.
- **Storage**: Local filesystem under Reflex's upload directory.
- **Transcoding**: `ffmpeg-python` builds command graphs; system `ffmpeg` does the actual conversion.

—

4. Core Modules and Responsibilities

4.1 `video_to_mp4/video_to_mp4.py`

- Defines the index page layout
- Renders:

- `upload_zone()`

- `job_list()`
- Registers route `/`

4.2 `video_to_mp4/states/app_state.py`

The central control plane:

- Holds UI flags (help tooltips, confirm dialog)
- Maintains `recent_jobs: list[FileJob]`
- Stages uploaded files to disk
- Spawns background conversion tasks using `@rx.event(background=True)`
- Tracks conversion progress by parsing FFmpeg `out_time_ms`

4.3 `video_to_mp4/components/upload_zone.py`

- Upload drop zone using `rx.upload.root(id="upload_box")`
- "Convert to MP4" triggers `AppState.open_confirm(rx.upload_files("upload_box"))`
- Confirm dialog calls:

- `AppState.confirm_upload()` to enqueue jobs and start background processing

- `AppState.close_confirm()` to cancel and delete staged files

4.4 `video_to_mp4/components/job_list.py`

- Renders recent jobs in a table
- Actions by status:

- Complete: download (anchor tag) + delete

- Error: retry + delete
- Other: delete (cancel)
- Download uses `rx.get_upload_url(job["converted_filename"])`

–

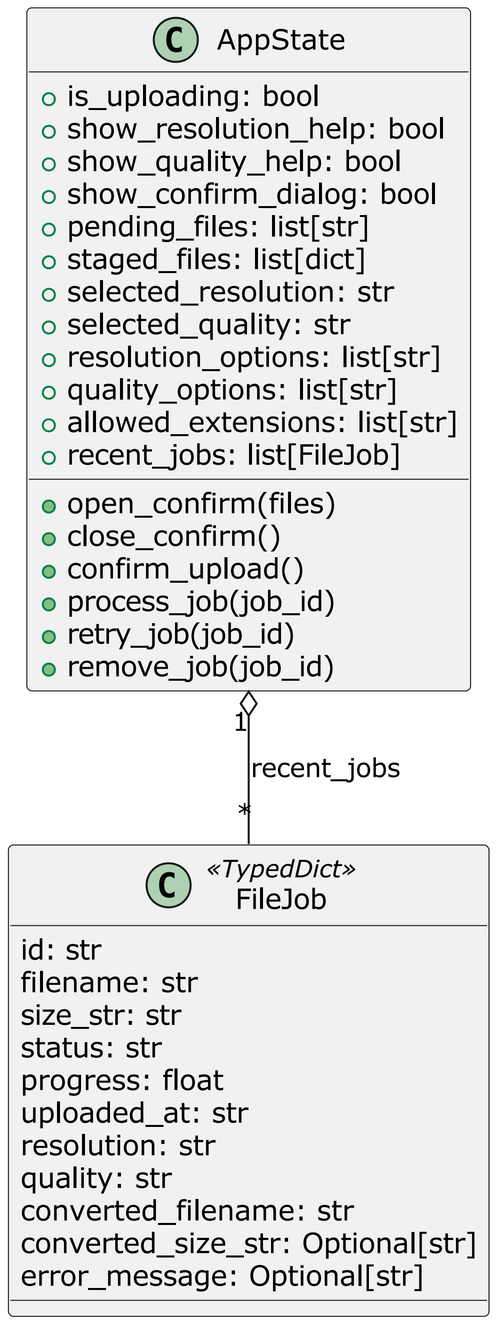
5. Data Design

5.1 Data Model: `FileJob`

`FileJob` is a `TypedDict` containing:

- `id`: str
- `filename`: str (stored filename in upload dir)
- `size_str`: str (human-readable input size)
- `status`: str (“Queued” | ”Processing” | ”Complete” | ”Error”)
- `progress`: float (0.0–100.0)
- `uploaded_at`: str (HH:MM)
- `resolution`: str (Original | 4K | 1080p | 720p | 480p)
- `quality`: str (Standard | High | Maximum)
- `converted_filename`: str
- `converted_size_str`: Optional[str]
- `error_message`: Optional[str]

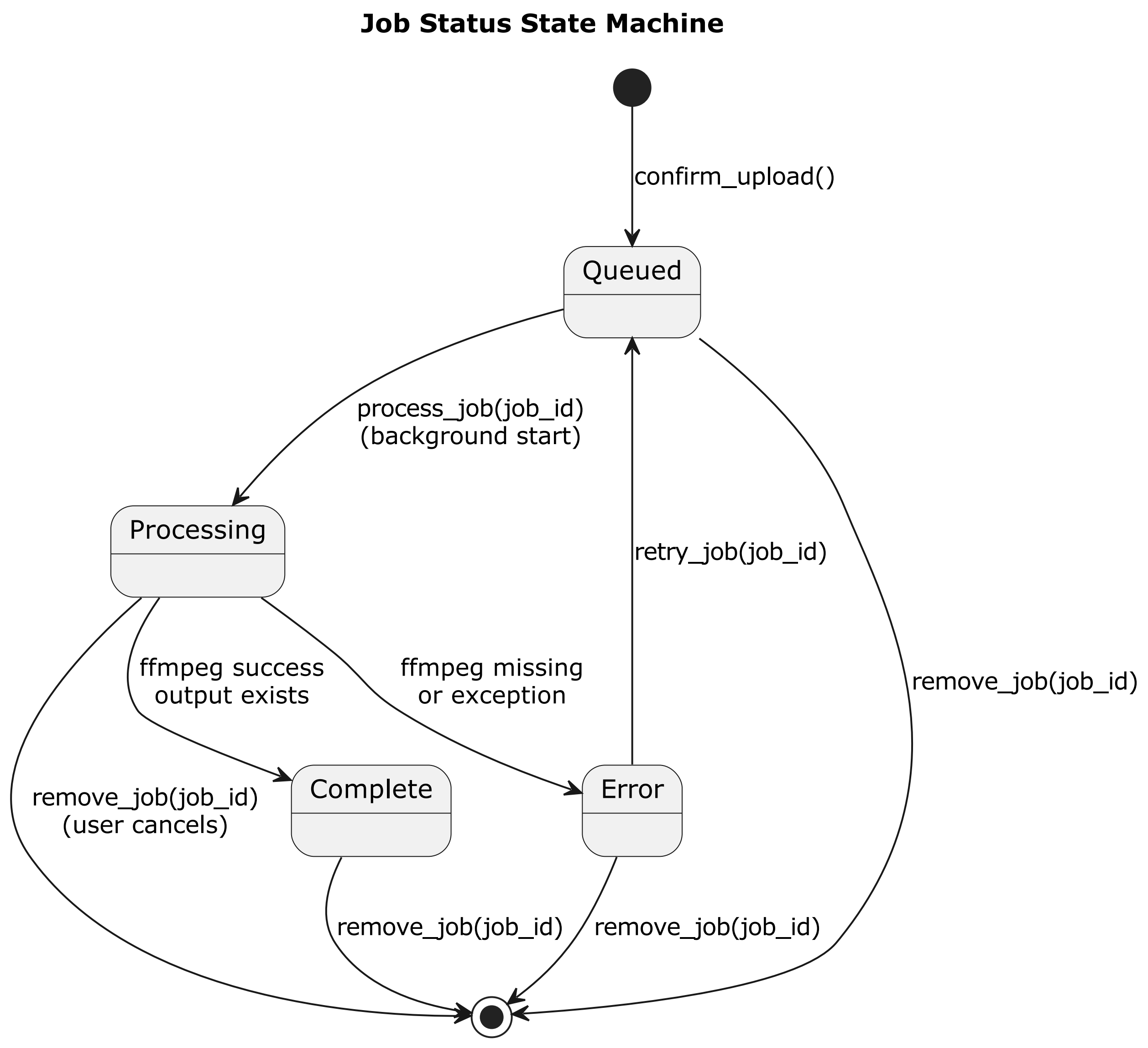
Data Model - FileJob and AppState



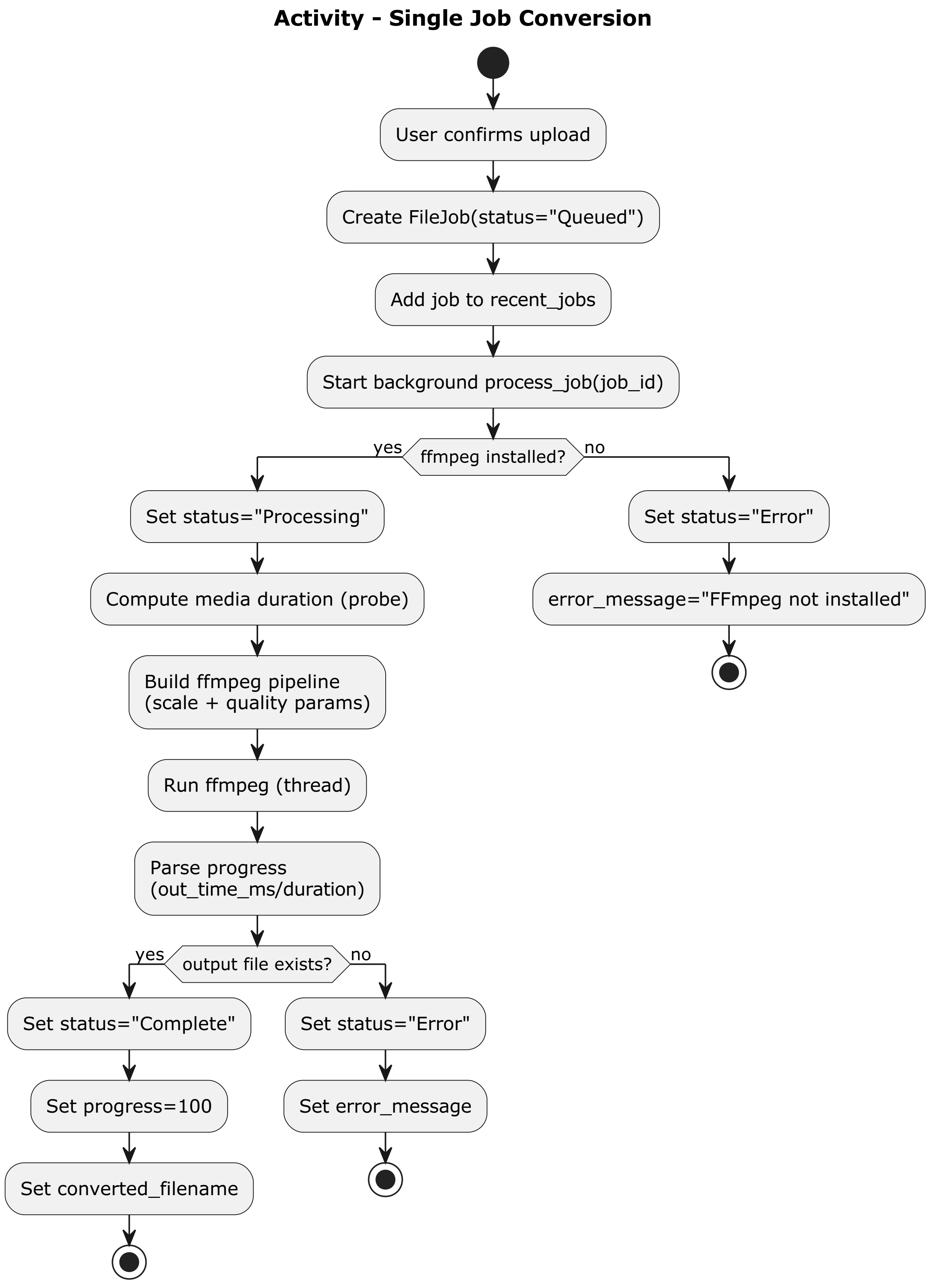
—

6. Job Lifecycle

6.1 Status State Machine



6.2 Activity Flow (Single Job)

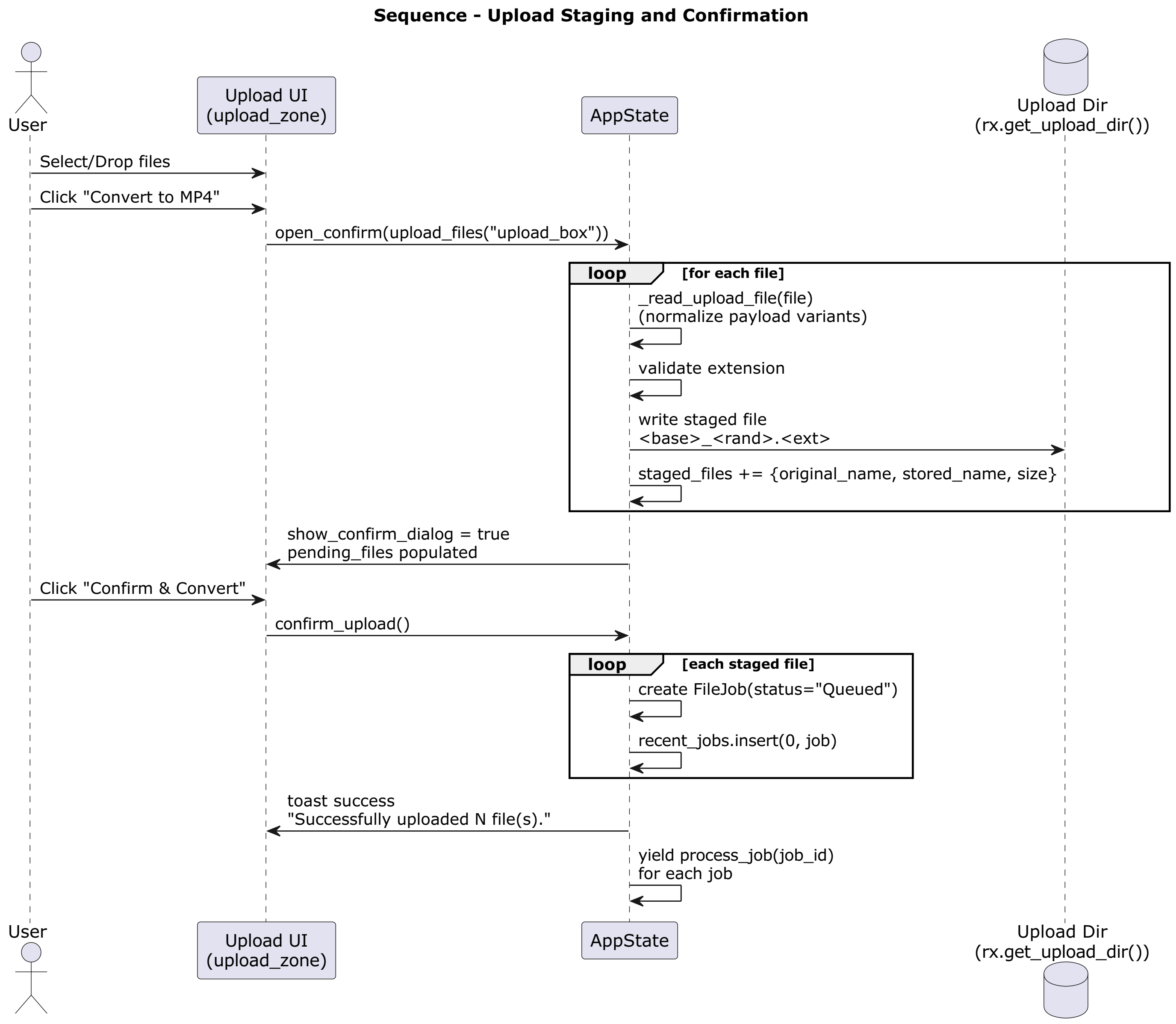


Activity - Single Job Conversion
User confirms upload
Create FileJob(status="Queued")
Add job to recent_jobs
Start background process_job(job_id)
ffmpeg installed?
yes
no
Set status="Processing"
Compute media duration (probe)
Build ffmpeg pipeline (scale + quality params)
Run ffmpeg (thread)
Parse progress (out_time_ms/duration)
output file exists?
yes
no
Set status="Complete"
Set progress=100
Set converted_filename
Set status="Error"
Set error_message
Set status="Error"
error_message="FFmpeg not installed"

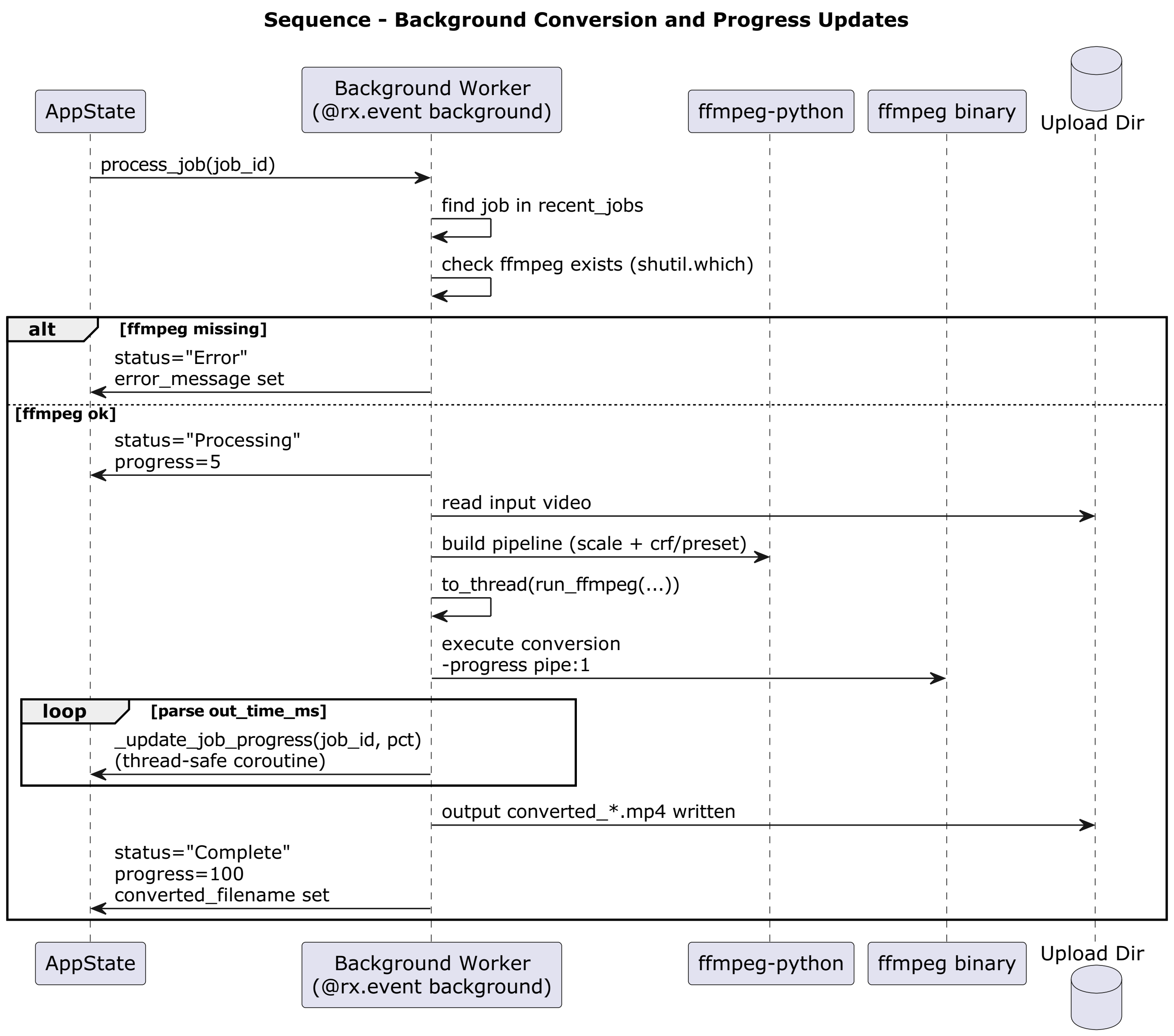
—

7. Detailed Workflow Sequences

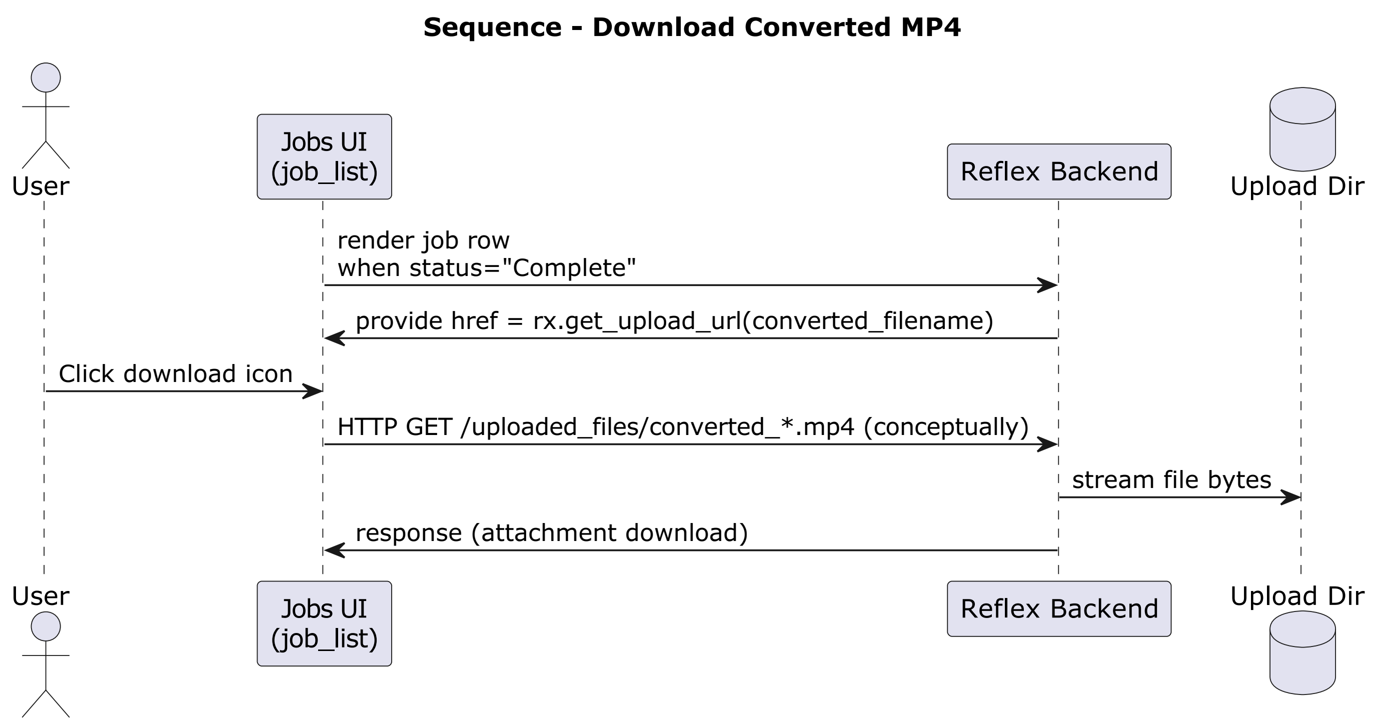
7.1 Upload → Confirm Dialog → Enqueue Jobs


Sequence - Upload Staging and Confirmation
User
Upload UI (upload_zone)
AppState
Upload Dir (rx.get_upload_dir())
Select/Drop files
Click "Convert to MP4"
open_confirm(upload_files("upload_box"))
loop [for each file]
_read_upload_file(file) (normalize payload variants)
validate extension
write staged file <base>_<rand>.<ext>
staged_files += {original_name, stored_name, size}
show_confirm_dialog = true pending_files populated
Click "Confirm & Convert"
confirm_upload()
loop [each staged file]
create FileJob(status="Queued")
recent_jobs.insert(0, job)
toast success "Successfully uploaded N file(s)."
yield process_job(job_id) for each job
User
Upload UI (upload_zone)
AppState
Upload Dir (rx.get_upload_dir())

7.2 Background Conversion and Progress Updates


Sequence - Background Conversion and Progress Updates
AppState
Background Worker (@rx.event background)
ffmpeg-python
ffmpeg binary
Upload Dir
process_job(job_id)
find job in recent_jobs
check ffmpeg exists (shutil.which)
alt [ffmpeg missing]
status="Error" error_message set
[ffmpeg ok]
status="Processing" progress=5
read input video
build pipeline (scale + crf/preset)
to_thread(run_ffmpeg(...))
execute conversion -progress pipe:1
loop [parse out_time_ms]
_update_job_progress(job_id, pct) (thread-safe coroutine)
output converted_*.mp4 written
status="Complete" progress=100 converted_filename set

7.3 Download Converted File


Sequence - Download Converted MP4
User
Jobs UI (job_list)
Reflex Backend
Upload Dir
render job row when status="Complete"
provide href = rx.get_upload_url(converted_filename)
Click download icon
HTTP GET /uploaded_files/converted_*.mp4 (conceptually)
stream file bytes
response (attachment download)

—

8. Conversion Design Details

8.1 Resolution Handling

In `process_job`:

- `Original`: no scale filter
- `4K`: `scale -1 2160`
- `1080p`: `scale -1 1080`
- `720p`: `scale -1 720`
- `480p`: `scale -1 480`

The `-1` preserves aspect ratio while fixing height.

8.2 Quality Handling (CRF + Preset)

Defaults:

- `crf = 23`, `preset = "medium"`

Modes:

- `Standard`: `crf = 28`, `preset = "fast"` (smaller file, lower quality)
- `High`: `crf = 18`, `preset = "slow"`

- `Maximum`: `crf = 15`, `preset = "veryslow"` (best quality, slowest)

Output codecs:

- Video: `libx264`
- Audio: `aac`

8.3 Progress Tracking Strategy

- Attempts to probe duration using `ffmpeg.probe(...)`
- Runs ffmpeg with `-progress pipe:1 -nostats`
- Parses `out_time_ms=...` lines and computes:

- `percent = min(99.99, max(0.0, elapsed / duration * 100))`

- Updates state via `asyncio.run_coroutine_threadsafe(...)`

This design avoids blocking the Reflex event loop and keeps UI responsive.

—

9. Error Handling

9.1 Validation Errors

- Invalid extension: produces toast errors during staging (`open_confirm`)
- No files selected: toast error and returns

9.2 Runtime Errors

- Missing `ffmpeg` binary: job becomes `Error` with message ”Server Error: FFmpeg not installed”
- Input file missing: `FileNotFoundError`
- Conversion fails: any exception stored as `error_message`

9.3 Cleanup

- Cancel confirm dialog (`close_confirm`) deletes staged files
- Remove job deletes:

- original staged input file

- converted file (if exists)

—

10. Security Considerations

10.1 File Type Validation

- Extension-based allowlist:
- `avi`, `mov`, `mkv`, `wmv`, `mp4`, `webm`

Recommendation (future hardening):

- Validate MIME type and/or inspect container format to reduce spoofing risks.

10.2 Path Safety

- Stored filenames are generated with random suffix and written under `rx.get_upload_dir()`.
- No user-controlled paths are joined directly (good baseline).

10.3 Multi-user Isolation (Not implemented)

Current jobs are in-memory and not scoped per user/session. In a shared deployment, users would see each other’s jobs.

Future options:

- Scope jobs to session/user id
- Separate upload directories per user
- Add auth layer

–

11. Performance Considerations

- Conversion is CPU-intensive and may saturate the server.
- Each job uses FFmpeg in a background thread; multiple jobs can run concurrently if multiple background events overlap.

Recommendations for scaling:

- Limit concurrency (e.g., semaphore)
- Add a job queue with worker count
- Offload to separate worker process/container

–

12. Deployment

12.1 Local Development

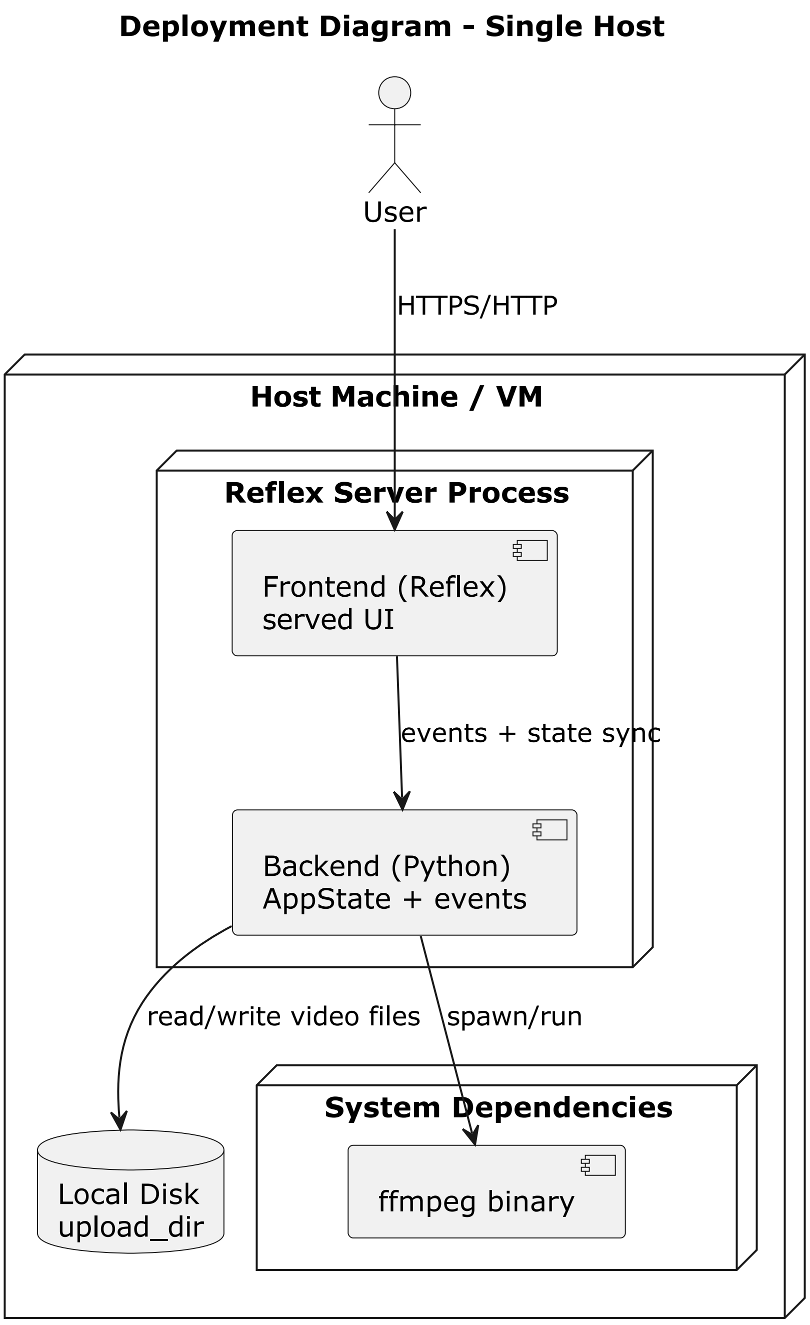
Prereqs:

- Python 3.11
- Poetry
- FFmpeg installed (macOS: `brew install ffmpeg`)

Run:

- `poetry run ./reflex_rerun.sh`
- Frontend: `http://localhost:3000`

12.2 Deployment Diagram (Conceptual)



—

13. Extension Points / Roadmap Ideas

13.1 Capacity Limit Feature (Planned)

`plan.md` mentions “100GB capacity limit” and there is a `capacity_indicator.py`, but `AppState` currently does not implement:

- MAX_CAPACITY_GB
- usage_percentage / used_capacity tracking

Suggested implementation approach:

- Compute `sum(file_size)` of all files in upload_dir (or per-session)
- Enforce max capacity at staging time
- Add cleanup policy (LRU) or user-managed deletion

13.2 Better Job Persistence

- Store job metadata in SQLite (or a small JSON DB)
- Persist across restarts
- Optionally persist original files until download

13.3 Worker Queue

- Add a conversion queue and worker pool
- Provide fair scheduling
- Improve stability under heavy load

—

14. Appendix: Key API/Event Surface (Backend)

14.1 User-facing Events

- `open_confirm(files)`
- `close_confirm()`
- `confirm_upload()`
- `process_job(job_id)` (background)
- `retry_job(job_id)`
- `remove_job(job_id)`

14.2 Helper Functions

- `_read_upload_file(file)` normalizes upload payload shapes
- `get_media_duration(path)` uses ffprobe via ffmpeg-python
- `run_ffmpeg(...)` executes conversion and optional progress parsing
- `_update_job_progress(job_id, pct)` updates job progress safely

—

15. Quick “How It Works” Summary

- UI collects files via Reflex upload component.

- Backend stages files into Reflex upload directory and shows a confirmation modal.
- Once confirmed, the system creates a job per file and starts background conversion.
- FFmpeg progress is parsed and pushed back into state; UI reflects progress live.
- When complete, UI shows a download action which serves the converted file from upload storage.

—

16. Precise Interaction Boundaries (Reflex Lifecycle, Upload Dir, Concurrency)

This section tightens the software design by explicitly modeling:
- Reflex frontend/backend lifecycle and event pipeline
- How `rx.get_upload_dir()` and `rx.get_upload_url()` define the storage/serving boundary
- Concurrency behavior of background conversions and UI actions
- Known race conditions in the current implementation, with recommended fixes

—

16.1 Reflex Execution Model and Event Pipeline

What the code actually does

- The UI triggers events via `on_click=AppState.open_confirm(rx.upload_files("upload_box"))`.
- `open_confirm` is an async event handler that:
- calls `upload_dir = rx.get_upload_dir()`
- writes staged files into that directory
- populates `pending_files` and `staged_files`
- toggles `show_confirm_dialog = True`
- `confirm_upload`:
- copies `staged_files` into a local list
- clears `pending_files`/`staged_files`
- creates `FileJob` entries (status `Queued`) in `recent_jobs`
- then yields `AppState.process_job(job_id)` for each job (scheduling background work)
- `process_job` is declared as `@rx.event(background=True)` and runs conversion logic:
- updates job status/progress while holding state lock (`async with self`)
- performs the heavy FFmpeg work in a worker thread via `await asyncio.to_thread(...)`

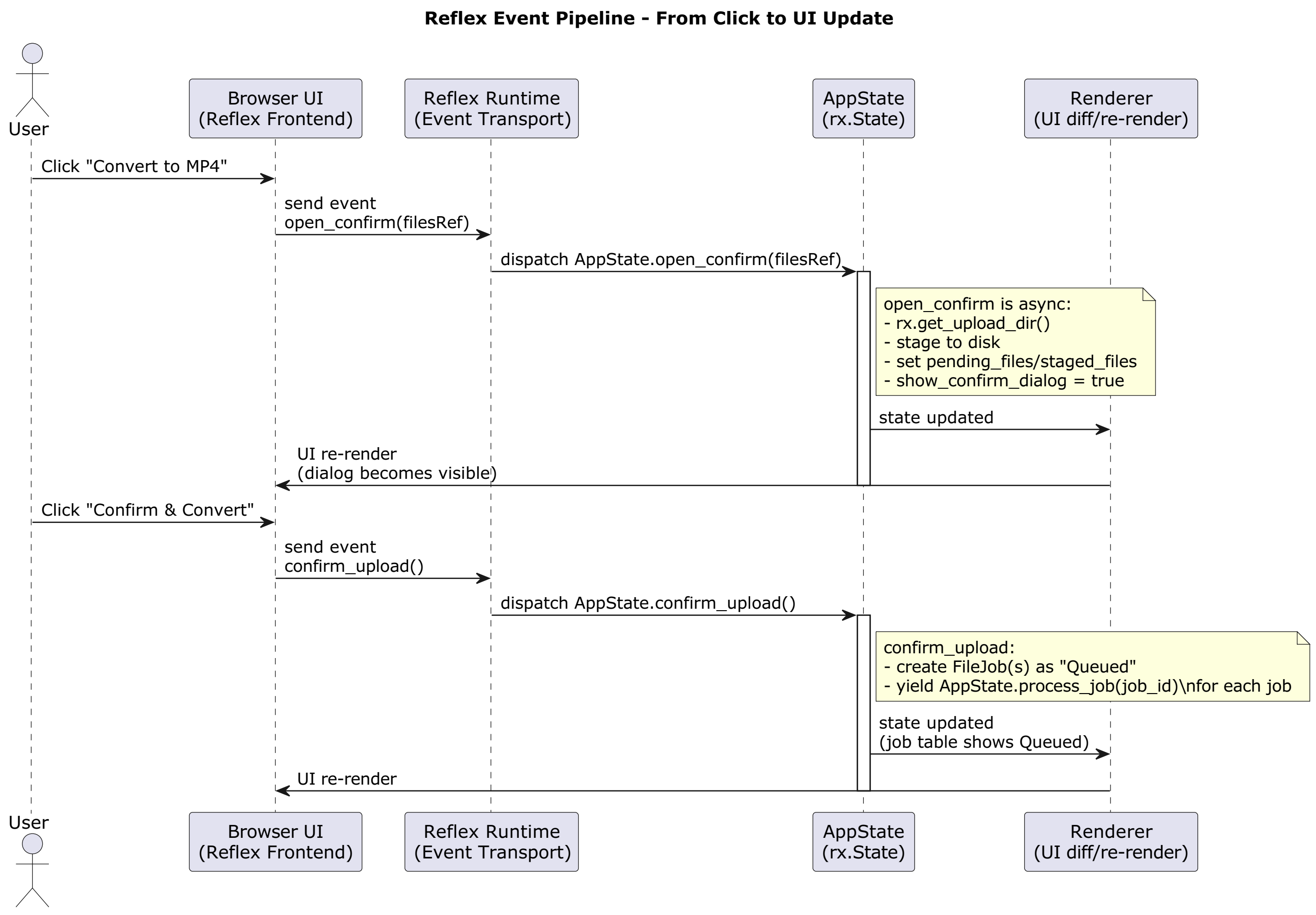
- streams progress back to the event loop using `asyncio.run_coroutine_threadsafe(...)`

Key boundary: State lock

- Any code under `async with self:` represents an **atomic state transaction** (serialized with other state updates).
- Heavy work intentionally happens **outside** that lock to avoid blocking other UI interactions.

—

16.2 Sequence: Reflex Frontend → Backend Event → State → UI Re-render


Reflex Event Pipeline - From Click to UI Update
User
Browser UI (Reflex Frontend)
Reflex Runtime (Event Transport)
AppState (rx.State)
Renderer (UI diff/re-render)
Click "Convert to MP4"
send event open_confirm(filesRef)
dispatch AppState.open_confirm(filesRef)
open_confirm is async:
- rx.get_upload_dir()
- stage to disk
- set pending_files/staged_files
- show_confirm_dialog = true
state updated
UI re-render (dialog becomes visible)
Click "Confirm & Convert"
send event confirm_upload()
dispatch AppState.confirm_upload()
confirm_upload:
- create FileJob(s) as "Queued"
- yield AppState.process_job(job_id)\nfor each job
state updated (job table shows Queued)
UI re-render

—

16.3 Upload Directory Boundary and URL Mapping

What the code uses

- Disk directory boundary:

- `upload_dir = rx.get_upload_dir()`

- staging writes: `upload_dir / unique_filename`
- conversion reads/writes: `upload_dir / input_filename` and `upload_dir / converted_*.mp4`
- Public URL boundary:

- downloads: `rx.get_upload_url(job["converted_filename"])`

Important: the concrete filesystem path and the public route are determined by Reflex runtime configuration.
The code does not hardcode them, which is good for portability.

How `_read_upload_file` searches for files

`_read_upload_file` supports multiple payload shapes, including:

- objects with `.read()`
- dict payloads with:

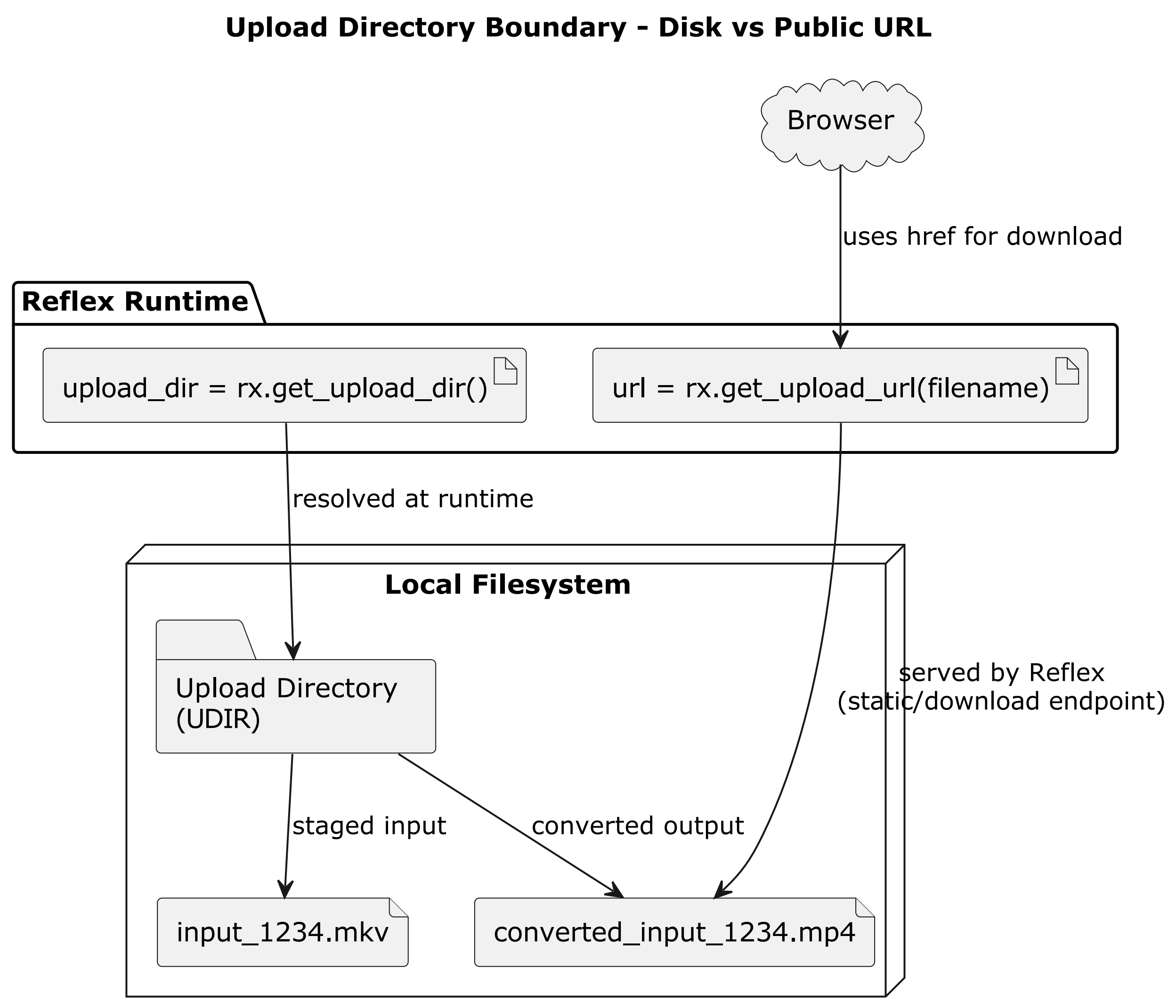
- inline `data`/`content`/`contents`

- inner `file` that can be bytes/list/base64/path/dict
- fallback `path` / `file_path` / `filepath`

It also tries several candidate directories when given a relative path:

- `rx.get_upload_dir() / path`
- `Path.cwd() / path`
- `Path.cwd() / ".web" / path`
- `Path.cwd() / ".web" / "public" / path`
- `Path.cwd() / ".web" / "backend" / path`
- `Path.cwd() / ".web" / "backend" / "uploaded_files" / path`
- `tempfile.gettempdir() / path`

This behavior is meant to tolerate differences in Reflex upload payloads across environments.



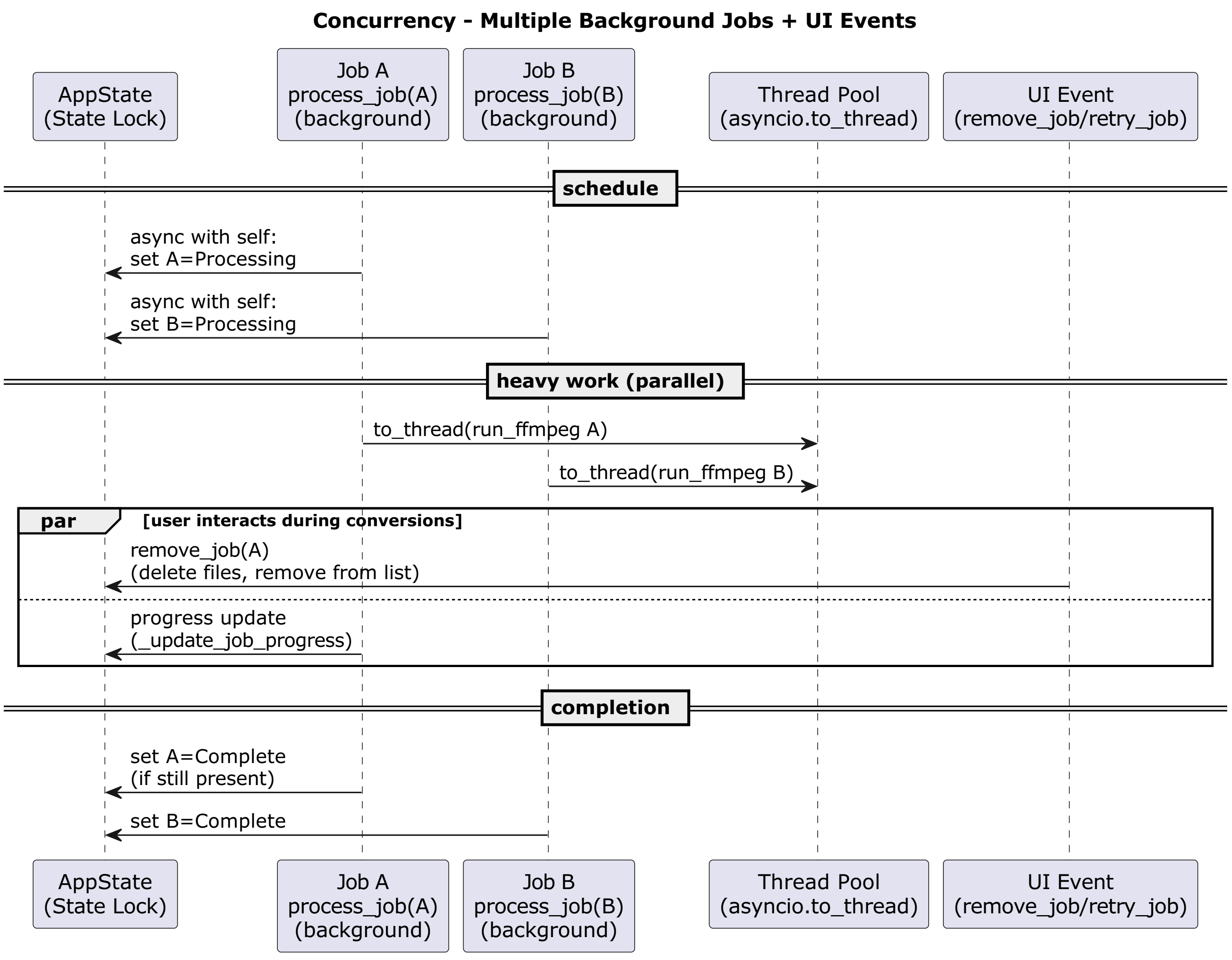
—

16.4 Concurrency Model: What Can Run at the Same Time

Observed in code

- Multiple jobs can be scheduled by `confirm_upload` in a loop:
- it yields `AppState.process_job(job_id)` for each job
- Each `process_job`:
- quickly takes the state lock to mark status
- then releases lock and runs FFmpeg in a worker thread (`asyncio.to_thread`)
- Therefore:
- **FFmpeg conversions can overlap** (multiple worker threads at once)
- UI events (remove/retry) can occur while conversions are running

Concurrency diagram


Concurrency - Multiple Background Jobs + UI Events
AppState (State Lock)
Job A process_job(A) (background)
Job B process_job(B) (background)
Thread Pool (asyncio.to_thread)
UI Event (remove_job/retry_job)
schedule
async with self: set A=Processing
async with self: set B=Processing
heavy work (parallel)
to_thread(run_ffmpeg A)
to_thread(run_ffmpeg B)
par
[user interacts during conversions]
remove_job(A) (delete files, remove from list)
progress update (_update_job_progress)
completion
set A=Complete (if still present)
set B=Complete

—

16.5 Race Conditions and Correctness Risks (Current Code)

This section documents **real risks** that follow directly from the implementation.

Risk 1: Index-based job updates can target the wrong job

In `process_job`, the code finds `job_idx` once (by scanning `recent_jobs`) and later updates using `self.recent_jobs[job_idx]`.
If another event modifies `recent_jobs` (e.g., `remove_job` removes a different job earlier in the list), the index may shift.

Impact:

- progress/status could be written into the wrong row
- completion could mark the wrong job as `Complete` / `Error`

Recommended fix:

- never rely on a stored list index across awaits
- re-locate the job by `job_id` each time you need to write, or maintain a dict keyed by id

Risk 2: Removing a job can delete files while FFmpeg is reading/writing

`remove_job` deletes:

- input file: `(upload_dir / job["filename"]).unlink(missing_ok=True)`
- output file (if present) If the conversion is in progress, deleting input can cause FFmpeg to fail mid-run.

Impact:

- conversion likely ends in error
- progress callback may still try to update state

Recommended fix:

- introduce a `canceled` flag per job and avoid deleting input until the worker acknowledges cancellation
- implement cooperative cancel: store process handle and send termination signal (advanced)
- simplest: disable delete button while `status == "Processing"` or show ”Cancel” that marks canceled and delays deletion

Risk 3: Output filename collisions across jobs

Output name: `converted_{Path(input_filename).stem}.mp4`.
Input filename includes a random suffix at staging time, so collision is unlikely, but not impossible if the same staged filename reappears.

Recommended fix:

- include `job_id` in output filename:

- `converted_{job_id}_{stem}.mp4`

—

16.6 Recommended Design Hardening (Minimal Changes)

Option A: Re-find job by id on every write (minimal invasive)

Replace index-based writes with a helper:

- `def _find_job_index(self, job_id) -> int`
- Use it inside every `async with self:` block, so updates always target the correct row.

Option B: Store jobs as a dict keyed by id (more robust)

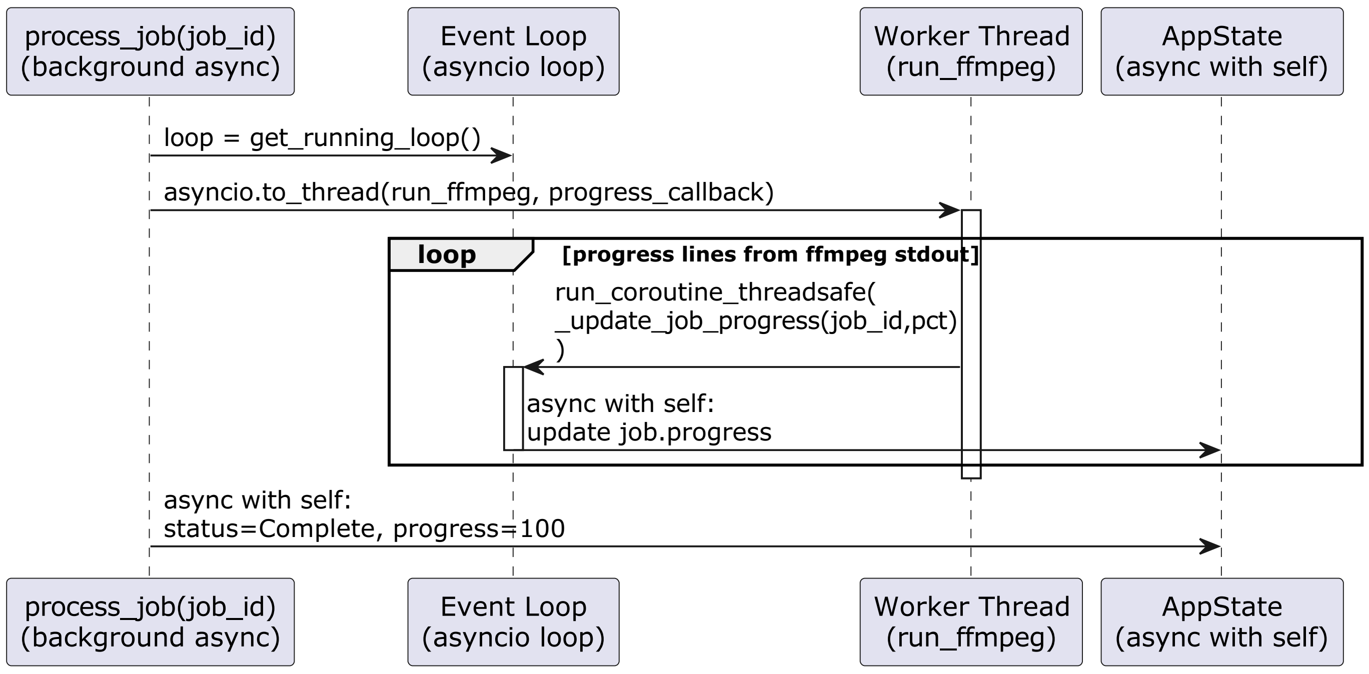
Change:

- `recent_jobs: list[FileJob]` To:
- `jobs_by_id: dict[str, FileJob]`
- and a separate `recent_job_ids: list[str]` for ordering.

—

16.7 Sequence: Progress Callback Thread → Event Loop → State Transaction

Sequence - Threaded FFmpeg Progress Updates into Reflex State

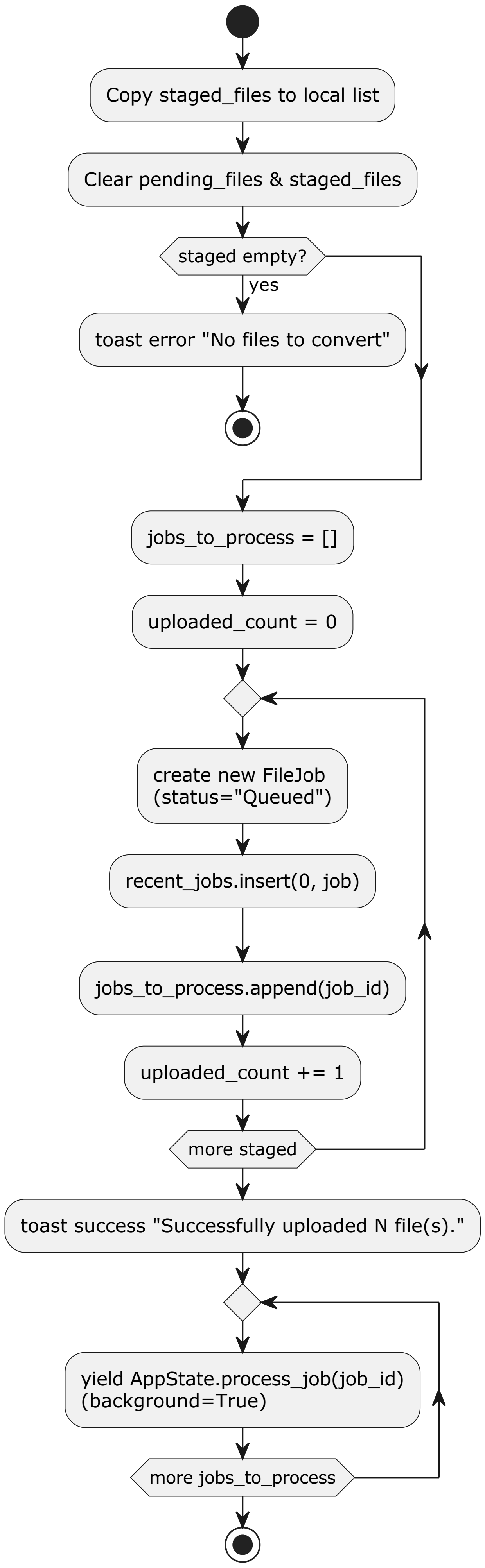

process_job(job_id) (background async)
Event Loop (asyncio loop)
Worker Thread (run_ffmpeg)
AppState (async with self)
loop = get_running_loop()
asyncio.to_thread(run_ffmpeg, progress_callback)
loop
[progress lines from ffmpeg stdout]
run_coroutine_threadsafe(_update_job_progress(job_id,pct))
async with self: update job.progress
async with self: status=Complete, progress=100

—

16.8 Activity: Multi-file Confirm & Batch Scheduling

This models exactly what `confirm_upload` does: create jobs first, then schedule background tasks.

Activity - confirm_upload() (Batch Scheduling)

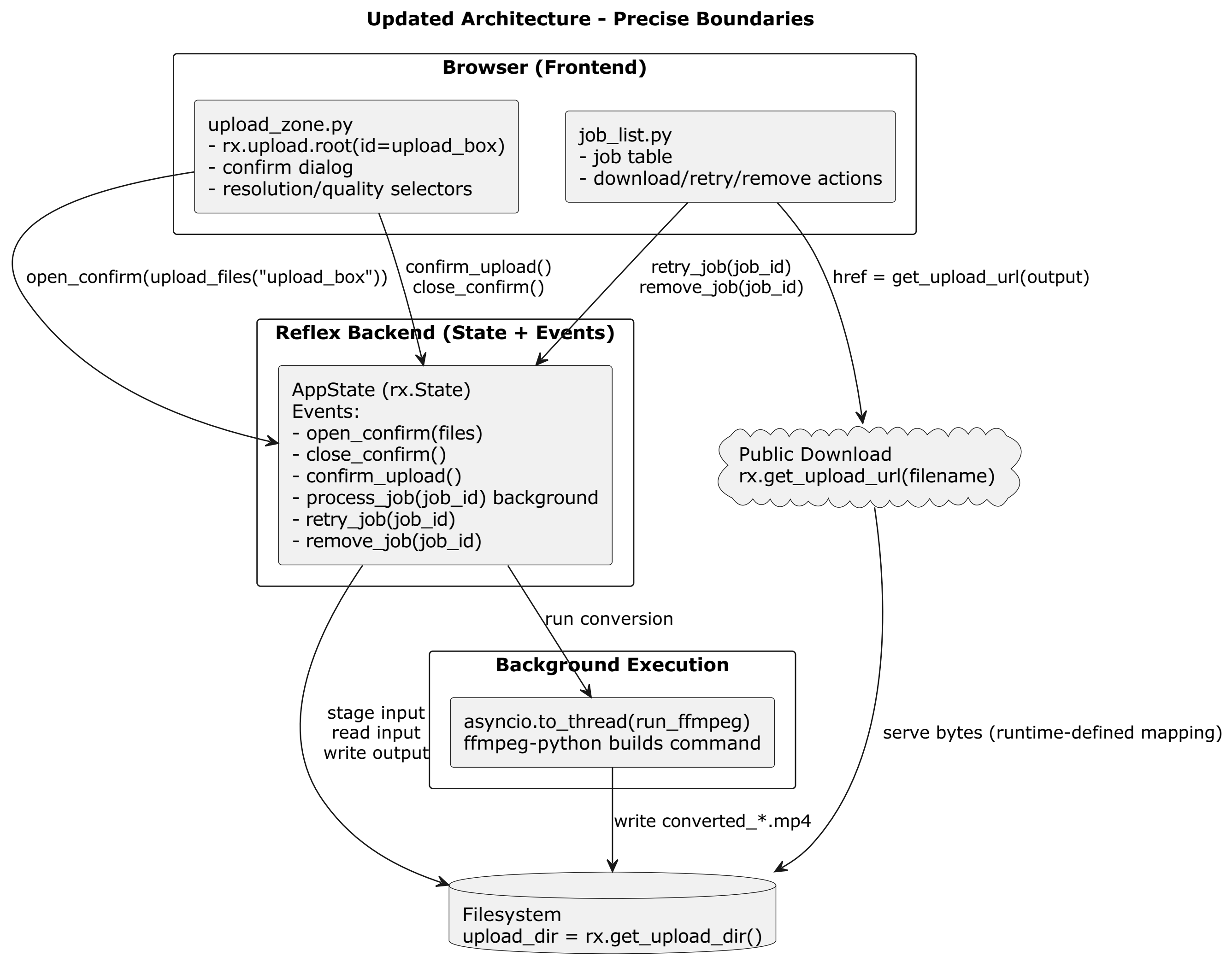

Copy staged_files to local list
Clear pending_files & staged_files
staged empty?
yes
toast error "No files to convert"
jobs_to_process = []
uploaded_count = 0
create new FileJob (status="Queued")
recent_jobs.insert(0, job)
jobs_to_process.append(job_id)
uploaded_count += 1
more staged
toast success "Successfully uploaded N file(s)."
yield AppState.process_job(job_id) (background=True)
more jobs_to_process

—

17. Precise Component Boundaries (Updated Architecture View)

This diagram explicitly separates:

- UI components (pure rendering)
- State machine (event handlers)
- Background worker (FFmpeg execution)
- Storage boundary (upload_dir)
- Public serving boundary (get_upload_url)


Updated Architecture - Precise Boundaries
Browser (Frontend)
upload_zone.py
- rx.upload.root(id=upload_box)
- confirm dialog
- resolution/quality selectors
job_list.py
- job table
- download/retry/remove actions
open_confirm(upload_files("upload_box"))
confirm_upload()
close_confirm()
retry_job(job_id)
remove_job(job_id)
href = get_upload_url(output)
Reflex Backend (State + Events)
AppState (rx.State)
Events:
- open_confirm(files)
- close_confirm()
- confirm_upload()
- process_job(job_id) background
- retry_job(job_id)
- remove_job(job_id)
Public Download
rx.get_upload_url(filename)
run conversion
Background Execution
asyncio.to_thread(run_ffmpeg)
ffmpeg-python builds command
stage input
read input
write output
serve bytes (runtime-defined mapping)
write converted_*.mp4
Filesystem
upload_dir = rx.get_upload_dir()

—

18. Test Implications (Concurrency-Aware)

Even without changing production code, the above boundaries define what should be tested.

18.1 Deterministic Unit Tests

- `_read_upload_file` with multiple payload shapes:

- object with `.read()`

- dict with `data` base64
- dict with `file: {path: ...}`
- `_format_size` formatting correctness
- `get_media_duration` fallback behavior when probe fails

18.2 Concurrency / Race Tests (Recommended)

- Start `process_job(jobA)` and then call `remove_job(jobB)` to verify jobA updates do not corrupt jobB (will likely fail under current index-based approach).
- Remove a job while its FFmpeg thread is running (should produce a controlled `Error` and not corrupt other jobs).

—

19. Summary of Precision Enhancements

- The design now explicitly models Reflex event transport, state locking, background execution, and filesystem/public-serving boundaries.
- The concurrency diagrams highlight real implementation risks (index drift, delete-while-processing).
- Recommended hardening options are provided with clear tradeoffs.

—

20. Minimal Patch Design: Race Condition Fix + Safe Cancel/Delete

This section provides a **minimal-invasive patch plan** to fix the real concurrency risks identified earlier:
1) Index drift when jobs are removed/reordered while background tasks update by list index

2) Deleting input/output files while FFmpeg is still reading/writing them

3) Output filename collisions / overwrites

Goals:
- Keep the current overall architecture (Reflex + AppState + upload_dir)
- Avoid introducing external systems (Redis/Celery)
- Change as little UI and state shape as possible
- Improve correctness under concurrent UI actions and background conversion

Non-goals:
- True hard cancellation of FFmpeg processes (requires managing process handles/signals)
- Multi-user isolation

–

20.1 Summary of Changes (Checklist)

Backend (`video_to_mp4/states/app_state.py`)

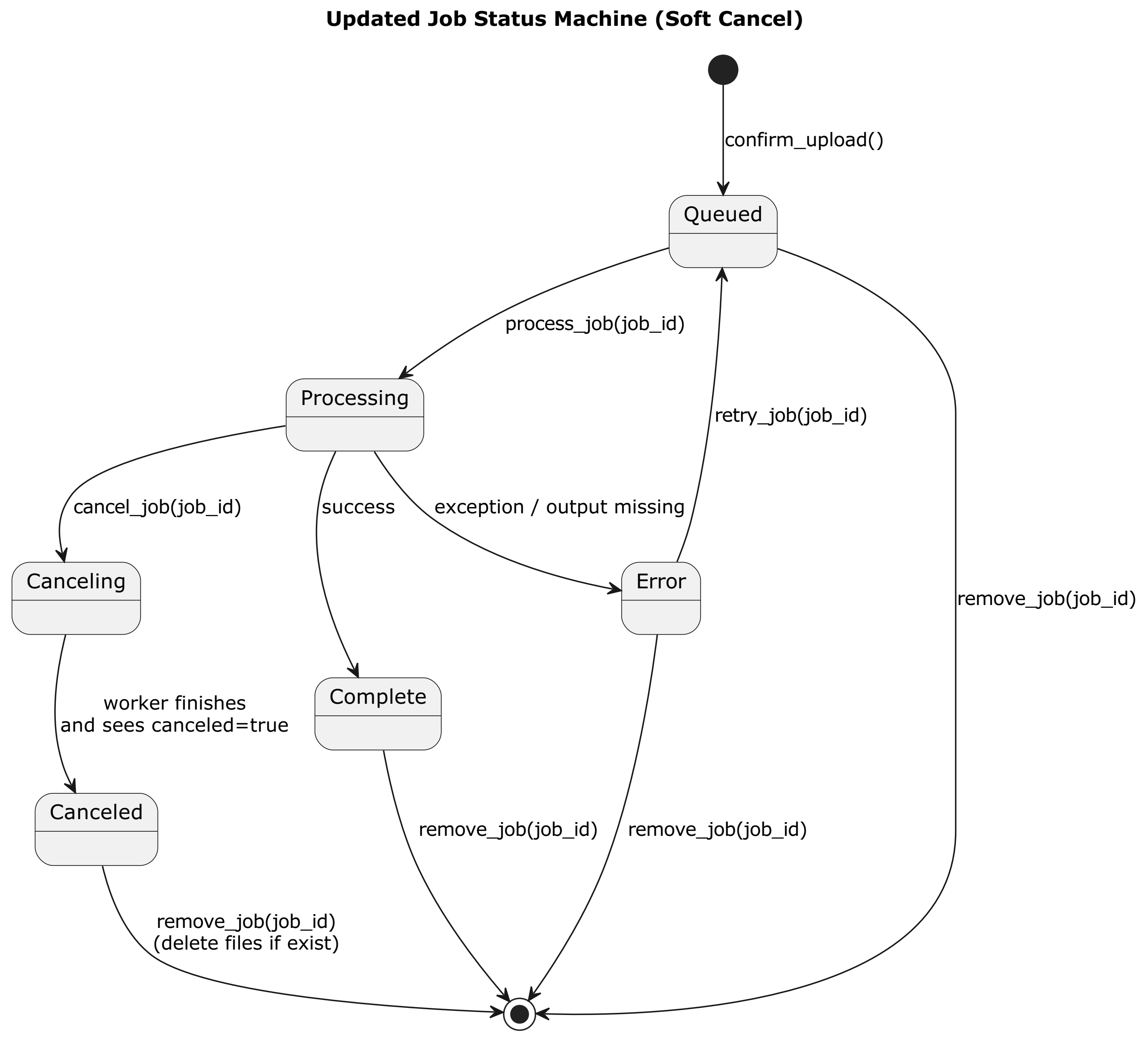
- [ ] Add `canceled: bool` field to `FileJob`
- [ ] Stop using stored list indices across awaits
  → Replace with ”find job by id” each time state is updated
- [ ] Make progress updates id-based and safe:
- If job no longer exists, ignore the update
- If job is canceled, ignore progress updates
- [ ] Implement **soft cancel**:
- For jobs in `Processing`, ”remove” becomes ”cancel” (no file deletion yet)
- After FFmpeg finishes, the worker checks `canceled` and cleans up safely
- [ ] Make output filename unique per job:
- `converted_{job_id}_{stem}.mp4`

Frontend (`video_to_mp4/components/job_list.py`)

- [ ] Disable “Delete” while `Processing`
- [ ] Show “Cancel” while `Processing`
- [ ] “Cancel” calls `AppState.cancel_job(job_id)` (new event)
- [ ] “Delete” remains for non-processing statuses

–

20.2 Updated Job State Machine (with Cancel)



Notes:

- “Canceling” is optional as a distinct status. If you want fewer strings, you can set `status="Processing"` and only toggle `canceled=true`, but having a visible `Canceling` improves UX clarity.

—

20.3 Patch Detail: Backend Implementation Plan

20.3.1 Data Model Change: Add `canceled`

Extend `FileJob` with:

- `canceled: bool` (default `False`)

```
# app_state.py
class FileJob(TypedDict):
    id: str
    filename: str
    size_str: str
    status: str
    progress: float
    uploaded_at: str
    resolution: str
    quality: str
    converted_filename: str
    converted_size_str: Optional[str]
    error_message: Optional[str]
    canceled: bool  # NEW
```

When creating a job in `confirm_upload()`:

```
job: FileJob = {
  ...
  "status": "Queued",
  "progress": 0.0,
  ...
  "error_message": None,
  "converted_size_str": None,
  "canceled": False,  # NEW
}
```

—

20.3.2 Core Fix: Stop Using Stored Indices Across Awaits

Add helper: find by id *inside the lock*

```
# app_state.py
def _find_job_index(self, job_id: str) -> int | None:
    for i, j in enumerate(self.recent_jobs):
        if j.get("id") == job_id:
            return i
    return None
```

Add helper: safe job update (id-based)

This ensures every update:

- locates the job fresh
- aborts if not found
- never writes to a stale index

```
# app_state.py
async def _update_job_fields(self, job_id: str, **fields) -> bool:
    async with self:
        idx = self._find_job_index(job_id)
        if idx is None:
            return False
        # If canceled, avoid updating progress/status unless explicitly allowed
        if self.recent_jobs[idx].get("canceled") and "status" not in fields:
            return False
        self.recent_jobs[idx].update(fields)
        return True
```

Replace all direct `self.recent_jobs[job_idx]...` writes

In `process_job`, instead of holding `job_idx` for the whole function, do:

- Early: `_update_job_fields(job_id, status="Processing", progress=5)`
- Progress callback: `_update_job_progress(job_id, percent)` uses `_update_job_fields`

—

20.3.3 Progress Updates: Ignore If Removed or Canceled

```
# app_state.py
async def _update_job_progress(self, job_id: str, percent: float) -> None:
    # clamp
    if percent < 0:
        percent = 0.0
    if percent > 99.99:
        percent = 99.99

    await self._update_job_fields(job_id, progress=percent)
```

If the job is removed while FFmpeg runs:

- `_find_job_index` returns None → update ignored (no corruption)

If the job is canceled:

- `_update_job_fields` ignores progress updates unless you intentionally allow them

—

20.3.4 Soft Cancel: Don’t Delete Files While Processing

New event: `cancel_job(job_id)`

```python
# app_state.py
@rx.event
def cancel_job(self, job_id: str):
    # Mark as canceled; do NOT delete files yet.
    for j in self.recent_jobs:
        if j.get("id") == job_id:
            j["canceled"] = True
            # Optional:
            j["status"] = "Canceling"
            j["error_message"] = None
            return
```

Update `remove_job(job_id)` behavior

If user tries to remove while processing, convert that request to cancel (safe):

```python
# app_state.py
@rx.event
def remove_job(self, job_id: str):
    upload_dir = Path(rx.get_upload_dir())

    for i, j in enumerate(self.recent_jobs):
        if j.get("id") != job_id:
            continue

        # If processing, do a soft cancel instead of deleting files
        if j.get("status") in ("Processing", "Canceling"):
            j["canceled"] = True
            j["status"] = "Canceling"
            return

        # Otherwise safe to delete immediately
        in_path = upload_dir / j["filename"]
        out_path = upload_dir / j.get("converted_filename", "")

        try:
            in_path.unlink(missing_ok=True)
        except Exception:
            pass

        try:
            if j.get("converted_filename"):
                out_path.unlink(missing_ok=True)
        except Exception:
            pass

        self.recent_jobs.pop(i)
        return
```

This prevents the most harmful condition: deleting the input while FFmpeg is still reading.

—

20.3.5 Worker Completion Logic: Cleanly Finalize Canceled Jobs

At the end of `process_job(job_id)`:

1. Re-check if job exists
2. Re-check canceled flag
3. If canceled → mark `Canceled` and cleanup files safely

Pseudo-structure:

```
# app_state.py (inside process_job)
# ... after ffmpeg finishes ...

async with self:
    idx = self._find_job_index(job_id)
    if idx is None:
        return
    canceled = self.recent_jobs[idx].get("canceled", False)

if canceled:
    # Mark as canceled and remove files (safe now - ffmpeg finished)
    await self._update_job_fields(job_id, status="Canceled", progress=0.0)
    self._cleanup_job_files(job_id)  # implement below (lock inside or careful)
    return
else:
    # Normal success path
    await self._update_job_fields(job_id, status="Complete", progress=100.0, ...)
```

Add a helper to avoid duplicating cleanup:

```
# app_state.py
def _cleanup_files_for_job(self, job: dict):
    upload_dir = Path(rx.get_upload_dir())
    try:
        (upload_dir / job["filename"]).unlink(missing_ok=True)
    except Exception:
        pass
    try:
        if job.get("converted_filename"):
            (upload_dir / job["converted_filename"]).unlink(missing_ok=True)
    except Exception:
        pass
```

When canceled:

- cleanup happens only after the background worker finishes (safe point)

—

20.3.6 Make Output Filename Unique Per Job

Replace:

- `converted_{stem}.mp4`

With:

- `converted_{job_id}_{stem}.mp4`

Example:

```
stem = Path(input_filename).stem
output_filename = f"converted_{job_id}_{stem}.mp4"
```

This avoids collisions and makes debugging easier.

—

20.4 UI Patch: Safe Buttons for Processing Jobs

In `video_to_mp4/components/job_list.py` update action rendering:

Rules:

- If status == Processing or Canceling:
- show “Cancel” button
- hide/disable “Delete”
- If status in (Queued, Error, Complete, Canceled):
- “Delete” is allowed
- If status == Error:
- show “Retry” + “Delete”
- If status == Complete:
- show “Download” + “Delete”

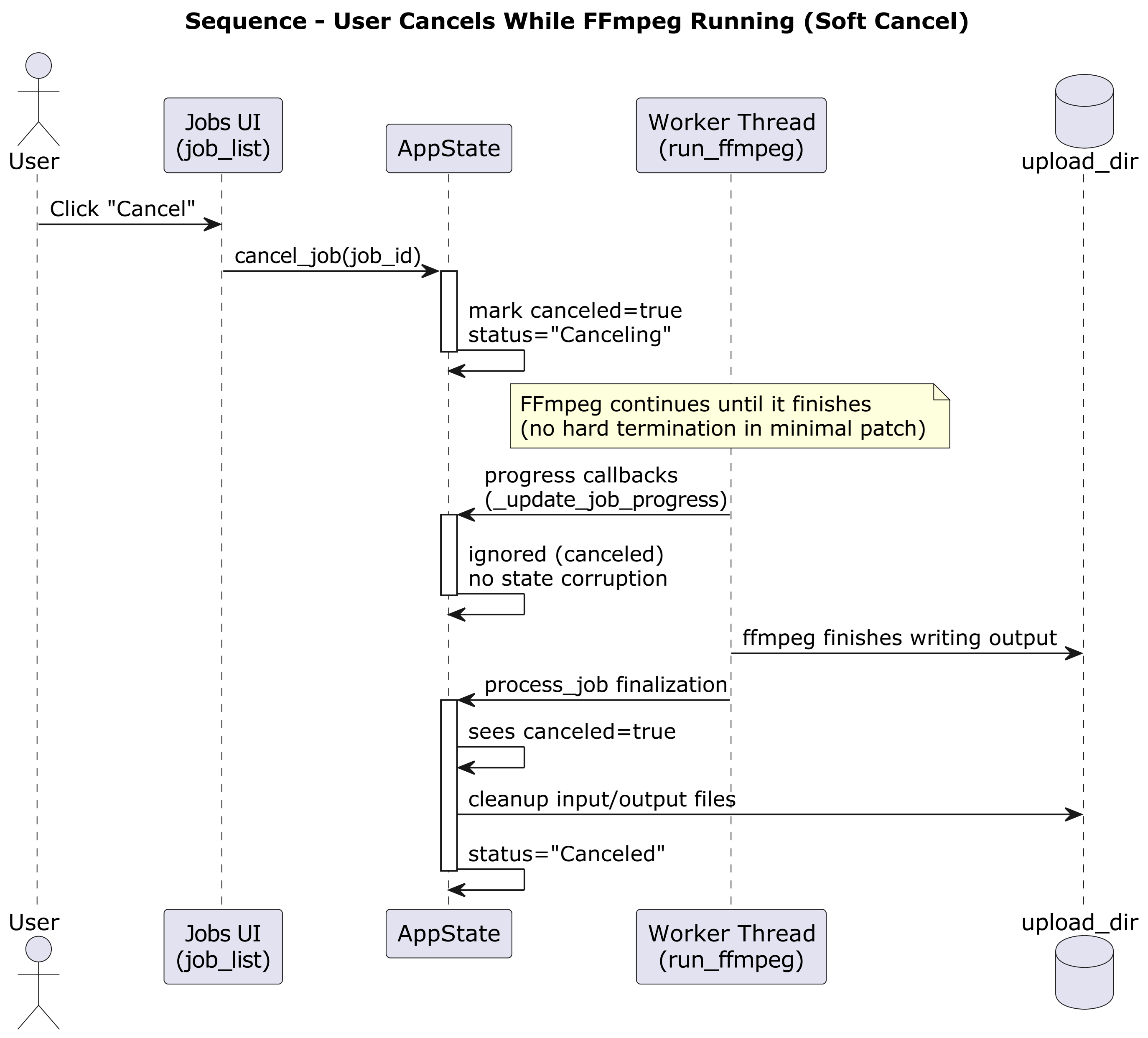
Pseudo-UI (conceptual):

```
def render_actions(job):
    status = job["status"]

    if status in ("Processing", "Canceling"):
        return rx.hstack(
            rx.button("Cancel", on_click=AppState.cancel_job(job["id"])),
            rx.text("Running...", size="1"),
        )

    if status == "Complete":
        return rx.hstack(
            rx.link("Download", href=rx.get_upload_url(job["converted_filename"])),
            rx.button("Delete", on_click=AppState.remove_job(job["id"])),
        )
    if status == "Error":
        return rx.hstack(
            rx.button("Retry", on_click=AppState.retry_job(job["id"])),
            rx.button("Delete", on_click=AppState.remove_job(job["id"])),
        )

    if status in ("Queued", "Canceled"):
        return rx.button("Delete", on_click=AppState.remove_job(job["id"]))
```

This is the minimal UI change that prevents unsafe deletion during processing.

—

20.5 Sequence: Cancel While Processing (Safe)



—

20.6 What This Patch Guarantees

After applying this patch plan:

- Background progress/status updates never corrupt the wrong job row (id-based lookup instead of stale list index)
- Users cannot trigger deletion of a file while FFmpeg is still processing it (remove → cancel during Processing)
- Output filenames become unique per job, avoiding accidental overwrites
- If a job disappears mid-run, progress updates are safely ignored

—

20.7 Optional Micro-Enhancements (Still Minimal)

- Add a small “Canceled” badge row and allow user to delete that entry
- Limit concurrent conversions with an `asyncio.Semaphore` to avoid CPU overload
- On cancel, set progress to 0 and keep status “Canceling” until worker finishes

—