# book2screenplay: A Hierarchical Multi-Agent Pipeline for Local Book-to-Screenplay and Narrated Video Adaptation

Shyamal Suhana Chandra
Crankshaft News
Pittsburg, Kansas
crankshaftnews@gmail.com

June 2026

**Abstract**

We present *book2screenplay*, a fully local pipeline that transforms a PDF book into a feature-length screenplay and a narrated 4K picture-video. The system orchestrates a hierarchy of large language model (LLM) agents—Director, Bible Builder, Scene Planner, parallel Writers, Continuity Editor, and Polish pass—backed by Ollama on consumer workstation hardware. Each stage emits structured JSON that downstream agents consume, enabling checkpointed, resumable execution over runs that span several hours. Beyond text, the pipeline generates per-scene cinematic stills, character portraits, macOS text-to-speech narration, and a Ken Burns–style 4K video with optional rich caption overlays. We report operational benchmarks on a 96 GB Mac Studio tuned to a ∼48 GB resident memory cap: roughly 70 scenes for a two-hour target, peak memory dominated by parallel ffmpeg encodes and 14B-parameter text models, and end-to-end wall-clock times of several hours. Key design findings include (1) role-specialized model sizing—14B for global structure, 8B for parallel scene writing—(2) explicit model eviction between stages to reclaim unified memory, (3) lenient JSON extraction with auto-close recovery for truncated LLM output, and (4) best-effort continuity repair that cannot invent facts absent from the Story Bible. We discuss limitations and directions for retrieval-augmented writing, human-in-the-loop review, and tighter cross-scene consistency.

## 1 Introduction

Adapting prose fiction into screenplays is a labour-intensive creative process. Professional adaptation requires distilling hundreds of pages into a ∼120-minute dramatic structure, maintaining character voice and continuity, and producing industry-standard formatted output. Recent advances in instruction-tuned LLMs make automated *drafting* of such adaptations plausible, but naïve single-prompt approaches fail on long books: context windows truncate source material, parallel generation introduces contradictions, and monolithic outputs are brittle to parse.

*book2screenplay* addresses these failures with a **staged, multi-agent architecture** that runs entirely on local hardware via Ollama [5]. The pipeline ingests a user-owned PDF, extracts and chunks text, builds a *Story Bible* (characters, locations, three-act beat sheet), plans ∼70 scene skeletons, writes scenes in parallel, runs a continuity review, polishes dialogue, and renders Fountain, LaTeX (`screenplay.cls`), and PDF artifacts. A second media track generates scene images, per-character portraits, macOS `say` narration, and a concatenated 4K MP4 with SRT sidecar subtitles.

This paper summarizes the system's design, the hierarchical algorithm, operational benchmarks observed during development, known limitations, and future work. The implementation is open source (Rust, Tokio async runtime) and targets a Mac Studio class workstation without cloud API dependencies.

# 2 Background

## 2.1 Screenplay Adaptation

Screenwriting conventions impose structure that differs from novelistic prose. Industry rule of thumb equates **one script page to one minute of screen time**; a two-hour feature therefore requires roughly 120 pages organized into three acts. Standard slug lines (INT./EXT. LOCATION — TIME), action lines in present tense, and dialogue blocks with parentheticals must be preserved for professional readers [1, 2].

Automated adaptation systems must therefore solve three coupled sub-problems: **(i)** global narrative compression (what to keep, merge, or cut), **(ii)** local scene expansion (dramatizing summary beats into playable scenes), and **(iii)** cross-scene consistency (names, timeline, voice, setting).

## 2.2 Local LLM Inference

Running models locally avoids recurring API cost, keeps proprietary manuscripts on-device, and enables long-running batch jobs. Ollama provides a HTTP `/api/generate` interface with model lifecycle controls (`keep_alive`, unload via zero keep-alive). On Apple Silicon, unified memory makes **peak resident set size (RSS)** the binding constraint rather than GPU VRAM alone.

We use:

- **qwen2.5:14b** for global reasoning (Director, Bible unification, Continuity, Scene Planner, Polish)—128K native context, strong JSON adherence.
- **llama3.1:8b** for parallel Writers—lower KV-cache footprint per concurrent slot.
- **x/z-image-turbo** (Ollama `x/` namespace) for scene stills and character portraits [6].

## 2.3 Related Work

Prior book-to-film pipelines in industry remain human-driven; academic work on LLM screenwriting [3, 4] focuses on short-form generation or interactive co-writing rather than end-to-end adaptation of full novels. Retrieval-augmented generation (RAG) over book embeddings is a natural extension but is not yet wired into v1 of this system (an `nomic-embed-text` hook exists in configuration for future use). Our contribution is an **integrated, checkpointed production pipeline** spanning text, image, speech, and video with explicit memory orchestration on a single machine.

# 3 System Architecture and Algorithm

## 3.1 Overview

Figure 1 summarizes the data flow. Every stage writes artifacts under `work/` and skips recomputation when outputs exist, making the pipeline crash-safe and resumable.

## 3.2 Stage 1: Ingestion and Chunking

PDF text is extracted and cached as `book.txt`. A paragraph-aware chunker splits prose into segments of at most $\sim$2500 approximate tokens (words $\times$1.3), preserving paragraph boundaries. Chunking bounds per-call context for Bible extraction while allowing the Director to consume a

Listing 1: Pipeline stages (conceptual).

```
book.pdf
  -> [Extract] -> raw text + paragraph chunks (~2500 tokens each)
  -> [Director] -> treatment: logline, themes, 24--40 beats (3 acts)
  -> [Bible Builder] -> parallel per-chunk extraction + merge/unify
  -> [Scene Planner] -> ~70 scene skeletons (goal / conflict / outcome
     )
  -> [Writers x N] -> parallel Fountain-structured scenes (JSON)
  -> [Continuity] -> patch list applied to scenes
  -> [Polish] -> per-scene dialogue refinement
  -> [Render] -> screenplay.fountain, screenplay.tex, screenplay.pdf
  -> [Images] -> one PNG per scene
  -> [Portraits] -> one PNG per speaking character (rich caption mode)
  -> [TTS] -> per-line WAV via macOS `say`
  -> [Video] -> 4K Ken Burns clips, concat, SRT + MP4 chapters
```

**digest**—first and last paragraph of each chunk, capped at 60 000 characters—for global shape without loading the entire book into one prompt.

### 3.3 Stage 2: Director and Story Bible

The **Director** agent (14B model, 65K context, JSON output) produces a treatment: title, logline, themes, tone, setting, time period, and an ordered beat list with estimated page counts summing to the target runtime (default 120 minutes). Act breaks (default pages 25 and 90) anchor three-act structure.

The **Bible Builder** runs *in parallel* over chunks, extracting characters (canonical name, aliases, role, voice, arc, relationships) and locations. Heuristic merge by normalized name deduplicates records; a final **unify** call trims to at most 20 named characters and normalizes contradictions against the treatment shell.

### 3.4 Stage 3: Scene Planning

The Scene Planner expands every beat into one or more **scene skeletons** with unique IDs (`S001`, ...), slug metadata (INT/EXT, location, time of day), participating characters, dramatic goal/conflict/outcome, and estimated page length (1–3 pages by default). Malformed skeleton records are dropped individually so a single JSON error does not abort the batch.

### 3.5 Stage 4: Parallel Writing

Each skeleton is assigned to an independent **Writer** agent (8B model, 16K context). Writers receive a *narrowed* bible containing only characters in that scene, reducing KV-cache pressure. Output is a JSON `Scene` with ordered elements (action, dialogue, parenthetical, transition, shot) plus an `image_prompt` for later illustration. Temperature increases slightly on JSON parse retries (up to `max_retries`).

Concurrency is limited by `writers.parallel` (default 3) and a global semaphore (`max_parallel`, default 14). Requests round-robin across multiple Ollama server instances on distinct ports for true parallelism.

### 3.6 Stage 5: Continuity and Polish

The **Continuity** agent scans the assembled screenplay in windows of eight scenes, emitting patches (`rename_character`, `fix_line`, `drop_element`, `note`) for factual inconsistencies—

unknown characters, alias mismatches, anachronisms, voice violations—without aesthetic rewrites. Patches are applied deterministically before a per-scene **Polish** pass refines dialogue and action.

### 3.7 Stage 6: Rendering

Scenes compile to Fountain and LaTeX via Tera templates (`screenplay.cls`). The configured engine (`tectonic`, `latexmk`, or `xelatex`) produces `screenplay.pdf`.

### 3.8 Stage 7: Multimedia Synthesis

Before image generation, all text models are **explicitly unloaded** to reclaim ∼10 GB. Scene images (1920×1080, cinematic style prefix) and optional character portraits (512×512 headshots) are generated through Ollama's evolving image API (base64 in `/api/generate` response). The image model is then unloaded before video encoding.

**TTS:** macOS `say` renders each dialogue/action line to WAV; `ffprobe` records durations for subtitle timing. Voices are assigned from gendered pools with fallback to `Alex`.

**Video:** Each scene becomes a 3840×2160 clip: Ken Burns zoom (`zoom_end`=1.15), burned-in captions (plain SRT or rich ASS with per-character colours, slug cards, lower-thirds, act-break titles, progress bar, corner portraits), then concatenation with optional MP4 chapter metadata. An `ffmpeg_sem` caps parallel encodes (default 4) because each 4K render consumes 4–6 GB RSS.

### 3.9 Memory Orchestration

The launcher script (`start_ollamas.sh`) sets: `OLLAMA_MAX_LOADED_MODELS=1`, `OLLAMA_NUM_PARALLEL=1`, `OLLAMA_KEEP_ALIVE=2m`, `OLLAMA_FLASH_ATTENTION=1`, `OLLAMA_KV_CACHE_TYPE=q8_0`. Two server instances (ports 11434/11435) typically hold one text model and one image model simultaneously under a ∼48 GB cap on 96 GB hardware.

### 3.10 Robust JSON Parsing

LLM outputs frequently include markdown fences or truncate mid-object. The shared `extract_json` routine locates the first `{` or `[`, tracks bracket nesting, and on truncation attempts auto-close of strings and containers before deserialisation—critical for long Scene Planner and Director responses.

## 4 Benchmarks and Operational Findings

Formal accuracy benchmarks against human-adapted screenplays were not run in v1; instead we document **operational characteristics** observed during development on a 96 GB Mac Studio (M-series, 28 CPU cores), throttled to ∼48 GB peak RSS via configuration.

### 4.1 Scale Targets

### 4.2 Memory Budget

Table 2 breaks down dominant costs. Video parallelisation is the largest single knob: reducing `video.parallel` from 4 to 2 saves ∼12 GB peak at the cost of longer mux time.

### 4.3 Ollama Pool Presets

### 4.4 Wall-Clock Time

End-to-end generation for a full two-hour pipeline requires **several hours** on the reference hardware—dominated by sequential LLM calls for planning/continuity, parallel but retry-prone

Table 1: Default pacing and output scale (two-hour target).

| Parameter | Default / observed |
|---|---|
| Target runtime | 120 minutes (≈120 pages) |
| Beats (Director) | 24–40 |
| Scenes (Planner) | ∼70 |
| Scene length | 1–3 pages each |
| Writer parallelism | 3 concurrent agents |
| Video encode parallelism | 4 concurrent ffmpeg jobs |
| Intermediate disk | ≥30 GB |
| Model download size | ∼15 GB (three models) |

Table 2: Resident memory by pipeline stage (default configuration).

| Stage | Dominant cost | Typical RSS contribution |
|---|---|---|
| Text agents | qwen2.5:14b + KV cache | ∼10 GB per loaded 14B model |
| Writers (parallel) | llama3.1:8b × N | ∼3–5 GB per slot (16K ctx) |
| Images | x/z-image-turbo | ∼4 GB |
| Portraits | same image model | ∼2 s each (cached) |
| Video mux | ffmpeg 4K zoompan | ∼6 GB per parallel encode |
| **Peak (2-instance pool)** | | **∼48 GB** |

writer JSON generation, ∼70 image generations (∼2 s each for portraits on z-image-turbo), TTS for thousands of lines, and 4K ffmpeg encodes. Exact distributions depend on book length, scene count, and retry rates; checkpointing amortises failures across restarts.

### 4.5 Quality Observations

- **Structure:** Beat sheets and act breaks produce readable three-act shape; scene skeletons enforce goal/conflict/outcome per unit.
- **Voice:** Narrow-bible prompting and Polish improve per-character diction but parallel writers still drift without Continuity patches.
- **Continuity:** Best-effort patching fixes naming and anachronisms; it *cannot* invent plot facts missing from the bible.
- **Visuals:** Scene prompts deliberately avoid proper names ("grizzled sea captain" not "Captain Ahab") to reduce text-in-image artifacts.
- **Speech:** macOS voice availability varies; missing voices fall back to `Alex`, reducing cast distinction.

## 5 Future Work

1. **Retrieval-augmented writing:** Embed chunks with `nomic-embed-text` and inject relevant passages into Writer prompts for fidelity to source prose.

2. **Human-in-the-loop gates:** Editable bible and beat sheet approval before expensive parallel writing; diff-aware regeneration of single scenes.

3. **Stronger continuity:** Global entity registry, coreference resolution, and constraint solvers beyond patch-based LLM review.

Table 3: Recommended Ollama instance counts vs. peak RAM.

| Command | Peak RAM | Notes |
|---|---|---|
| `start_ollamas.sh 1` | ∼30 GB | Text/image swap; slowest |
| `start_ollamas.sh 2` | ∼48 GB | Default; no model swapping |
| `start_ollamas.sh 4` | ∼80 GB | Higher writer throughput |

4. **Evaluation harness:** Automated metrics (character recall, scene coverage vs. beats) and human rubrics against professional adaptations.
5. **Cross-platform TTS:** Replace macOS-only `say` with portable neural TTS (e.g. Piper, Coqui) for Linux/Windows deployments.
6. **Image API stabilisation:** Consolidate Ollama image endpoints as the `x/` model namespace matures; support higher-quality `x/flux-2-klein` as a quality tier.
7. **Licensing workflow:** Explicit rights attestation and provenance metadata in output artifacts.

## 6 Conclusion

*book2screenplay* demonstrates that a **hierarchically decomposed, locally executed multi-agent pipeline** can draft feature-scale screenplays and companion narrated videos from full-length books without cloud inference. Success depends less on any single model call than on **structure** (bible, beats, skeletons), **parallelism with reconciliation** (writers + continuity), **operational resilience** (checkpointing, lenient JSON parsing), and **memory-aware orchestration** (model eviction, capped ffmpeg concurrency). The system is a practical baseline for automated adaptation research; meaningful quality gains likely require retrieval grounding, interactive human editing, and rigorous evaluation—directions we identify for future releases.

## Acknowledgments

The author thanks the Ollama project and the open-weight model communities behind Qwen, Llama, and the `x/` image models for enabling fully local experimentation.

## References

[1] Syd Field. *Screenplay: The Foundations of Screenwriting.* Delta, revised edition, 2005.

[2] Robert McKee. *Story: Substance, Structure, Style, and the Principles of Screenwriting.* ReganBooks, 1997.

[3] Piotr Mirowski et al. Co-writing screenplays and theatre scripts with language models: An evaluation by industry professionals. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023. Representative work on LLM-assisted screenwriting.

[4] Atsuyuki Miyai et al. Can large language models write screenplays? 2024. Representative benchmark-oriented screenwriting study.

[5] Ollama. Ollama: Get up and running with large language models locally. https://ollama.com/, 2024. Accessed June 2026.

[6] Ollama. Image generation in ollama. https://ollama.com/blog/image-generation, 2025. Accessed June 2026.