My Weird Prompts

A Voice-to-Podcast Pipeline

Technical Architecture White Paper

Daniel Rosehill

February 2026 — Version 1.0

Pipeline V4 – Chatterbox TTS on Modal

Abstract

My Weird Prompts is a fully automated podcast pipeline that transforms short voice recordings into polished, multi-voice podcast episodes. A user records a question or topic on their phone; within 15–20 minutes a complete episode is published with AI-generated dialogue between two fictional hosts, original cover art, loudness-normalised audio, show notes, and an RSS feed entry.

The system is built on a serverless architecture using Modal for GPU compute, Cloudflare R2 for object storage, Neon PostgreSQL for metadata, and Vercel for static site hosting. At current scale the pipeline costs approximately $0.33–0.45 per episode in compute.

This document describes the production architecture as of February 2026, including the 12-stage pipeline, the two-pass editing system, the parallel TTS strategy, the safety and fault tolerance mechanisms, and the cost model.

> **Project Links**
> Website: myweirdprompts.com GitHub: MWP-Backend Recorder: recorder.myweirdprompts.com

Contents

Contents

1 Introduction

1.1 The Concept

My Weird Prompts (MWP) is an experiment in full-stack AI automation: a podcast where every episode begins with a single voice memo and ends as a published, multi-voice audio show — with no manual editing in between.

The show features two AI hosts:

- **Corn** — a relaxed, knowledgeable sloth who leads the conversation
- **Herman** — an enthusiastic, curious donkey who asks follow-up questions

Prompts are typically submitted by Daniel Rosehill (the show's producer) via a mobile Progressive Web App. The pipeline transcribes the prompt, researches the topic, writes a full dialogue script, generates cover art, synthesises speech with cloned voices, assembles a broadcast-ready episode, and publishes it to the web and podcast platforms.

1.2 Design Goals

The pipeline was designed around several principles:

1. **Zero human editing** — every stage is automated, from transcription to publication.
2. **Broadcast-quality output** — loudness-normalised to EBU R128 (-16 LUFS), with proper intro/outro, disclaimer, and credits.
3. **Fail-open safety** — non-critical failures (cover art, polish pass) degrade gracefully rather than aborting the episode.
4. **Cost efficiency** — the entire pipeline runs for under $0.50 per episode, using commodity T4 GPUs and free-tier AI APIs where possible.
5. **Full observability** — progress tracking, email notifications, job queuing, and recovery storage for failed episodes.

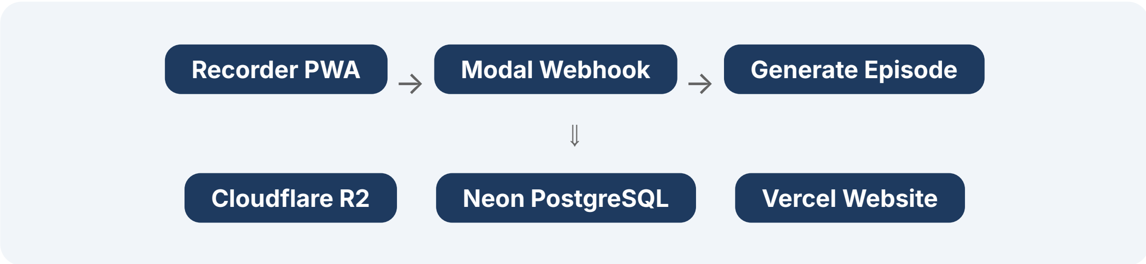
1.3 The Cast

Character	Description
Corn (Host)	A laid-back sloth with deep knowledge across topics. Leads conversations with measured insight and dry humour.
Herman (Co-host)	An energetic donkey who asks the questions listeners are thinking. Brings enthusiasm and follow-up curiosity.
Daniel (Producer)	The human behind the curtain. Submits voice prompts and maintains the pipeline. Occasionally acknowledged by the hosts.

2 System Architecture

2.1 High-Level Flow

The production system consists of four deployed components connected by webhooks and shared storage:



2.2 Deployment Topology

Component	Platform	URL
Recorder PWA	VPS (Docker)	recorder.myweirdprompts.com
Pipeline Webhook	Modal (serverless)	modal.run/.../webhook/generate
TTS Workers	Modal (T4 GPUs)	Internal (parallel workers)
Frontend Website	Vercel (SSG)	myweirdprompts.com
Admin CMS	Vercel (Next.js)	admin.myweirdprompts.com
Object Storage	Cloudflare R2	episodes.myweirdprompts.com
Database	Neon PostgreSQL	Serverless Postgres
Archival Storage	Wasabi S3	EU-Central-2 bucket

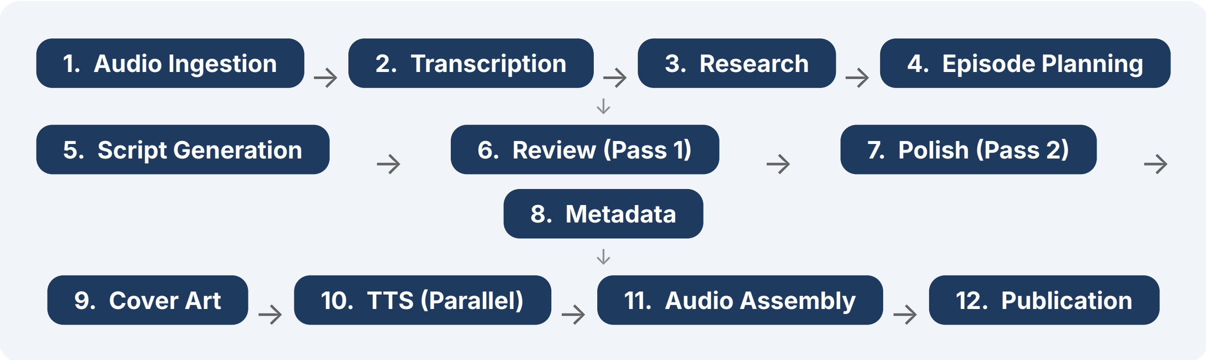
2.3 Infrastructure Stack

The pipeline uses exclusively serverless and managed services, with no dedicated servers beyond the recorder VPS:

- **Compute**: Modal (serverless containers with GPU scheduling)
- **Storage**: Cloudflare R2 (S3-compatible, zero egress fees), Wasabi (archival)
- **Database**: Neon PostgreSQL (serverless, auto-scaling)
- **Hosting**: Vercel (static site generation from Astro)
- **CI/CD**: GitHub Actions (auto-deploy on push to main)
- **DNS/CDN**: Cloudflare (custom domains, caching)

3 Pipeline Stages

Each episode passes through 12 stages. The full pipeline runs in a single Modal container (orchestrator) that spawns GPU workers for TTS. Total wall-clock time is typically 15–20 minutes.



3.1 Stage 1: Audio Ingestion & Validation

The pipeline receives an audio URL (typically from the Recorder PWA via Cloudflare R2) and performs initial validation:

- **Download**: HTTP GET with 120-second timeout and retry with exponential backoff
- **Size check**: Files under 1 KB are rejected as invalid
- **Format support**: MP3, WAV, WebM, OGG, FLAC, AAC, M4A (max 50 MB)

The audio is saved to a Modal shared volume for processing.

3.2 Stage 2: Transcription

The raw audio is transcribed using **Google Gemini's multimodal API** (model: `gemini-2.5-flash`). Rather than a pure speech-to-text service, Gemini listens to the audio and produces a cleaned transcript:

- Removes filler words (um, uh, like, you know)
- Eliminates false starts and repetitions
- Preserves core meaning, tone, and intent
- Supports disambiguation hints for technical terms

This multimodal approach captures nuances that pure ASR misses — tone, emphasis, and context.

3.3 Stage 3: Research Coordination

A lightweight research coordinator (also `gemini-2.5-flash`) analyses the transcript to determine if the topic references current events:

- Extracts key topics and entities
- Classifies whether web search is needed
- Generates focused search queries for logging

Actual web search is deferred to the script generation stage, where Gemini's **Google Search grounding** feature fetches real-time information inline.

3.4 Stage 4: Episode Planning

A dedicated planning agent (`gemini-2.5-flash`) creates a structured episode outline before script generation:

- **Segment breakdown** with specific points to cover
- **Key facts** and data to incorporate
- **Misconceptions** to address
- **Cross-episode references** from the episode memory system
- **Tone and pacing guidance**

The plan is formatted as a structured prompt section that the script generator follows as a roadmap. This produces more coherent, well-structured episodes than unguided generation.

The planning agent fails open — if it returns invalid JSON or errors, the pipeline continues without a plan.

3.5 Stage 5: Script Generation

The core creative step. Uses **Gemini 3 Flash Preview** (`gemini-3-flash-preview`) with multi-modal input:

- **Original audio** is passed alongside the text prompt, enabling the model to perceive tone, emphasis, and intent
- **Google Search grounding** is enabled for real-time fact-checking
- **Episode plan** provides the structural roadmap
- **Episode memory** includes the 3 most recent episodes for cross-references
- **Date context** ensures the model uses the correct current date

The target output is a diarized dialogue script (3,750 words / 25 minutes) in the format:

```
CORN: [dialogue text]
HERMAN: [dialogue text]
```

Key parameters: `max_tokens=8000` , `temperature=0.8` .

> **Why multimodal?** Passing the original audio rather than just the transcript lets the model pick up on enthusiasm, hesitation, or sarcasm that text transcription flattens. This produces more contextually appropriate responses.

3.6 Stage 6: Script Review (Pass 1)

The first of two editing passes, using **Gemini 3 Flash Preview** with **Google Search grounding** enabled:

- **Fact-checking**: Verifies claims against live web sources
- **Plan adherence**: Ensures all planned segments are covered

- **Depth check**: Adds substance where the script is thin
- **TTS compliance**: Fixes formatting that would confuse text-to-speech

The review agent receives the full script, original transcript, and episode plan. It returns the edited script as raw text (no JSON wrapping).

Safety mechanisms:
- **Shrinkage guard**: Rejects edits that reduce the script by more than 20%
- **Minimum length**: Rejects output under 1,000 characters
- **Fail-open**: Returns the original script if anything goes wrong

Parameters: `temperature=0.4`, `max_tokens=10000`.

3.7 Stage 7: Script Polish (Pass 2)

A lighter second pass using **Gemini 2.5 Flash** (no grounding needed):

- **Verbal tic removal**: Reduces overuse of "Exactly", "Absolutely", "That's a great point"
- **Sign-off cleanup**: Ensures no questions or new topics after goodbye
- **Flow improvement**: Smooths transitions and pacing
- **TTS final check**: Catches remaining formatting issues

This pass does **not** change facts or substance — only dialogue naturalness.

Safety mechanisms:
- **Shrinkage guard**: Rejects output if script shrinks by more than 15%
- **Fail-open**: Returns the original script on any error

Parameters: `temperature=0.3`, `max_tokens=10000`.

3.8 Stage 8: Metadata Generation

Uses **Gemini 2.5 Flash** to generate episode metadata from the final script:

- **Title**: Concise, engaging episode title
- **Slug**: URL-safe identifier
- **Description**: 2–3 sentence summary
- **Excerpt**: One-line teaser (for social media)
- **Tags**: Dynamic taxonomy from a registry of canonical tags
- **Category/Subcategory**: Hierarchical classification
- **Image prompt**: Description for cover art generation
- **Embedding**: Semantic vector for similarity search

Tags are generated using a taxonomy-aware system that maintains consistency across episodes and prevents tag sprawl.

3.9 Stage 9: Cover Art Generation

Uses **Fal AI** (`fal-ai/flux/schnell`) to generate a unique cover image:

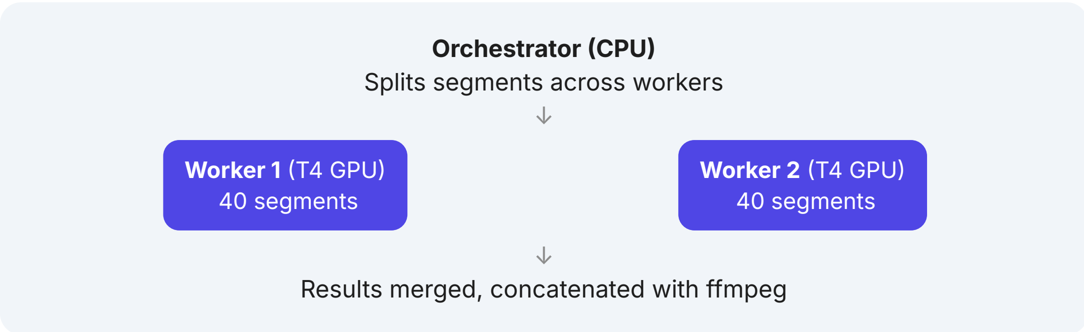
- Model receives the image prompt from metadata generation
- Generates one cover art variant
- Image uploaded to Cloudflare R2

Cover art is **non-critical** — if generation fails, the pipeline continues with a default cover image (graceful degradation).

3.10 Stage 10: Text-to-Speech (Parallel GPU Workers)

The most compute-intensive stage. Uses **Chatterbox TTS** (regular, not Turbo) running on **Modal T4 GPUs**.

3.10.1 Architecture



3.10.2 Key Optimisations

1. **Pre-computed voice conditionals**: Voice embeddings are computed once and cached in R2, eliminating 5–10 seconds of processing per segment.
2. **Parallel workers**: Segments are distributed across 2 GPU workers (configurable). Each worker loads the model once and processes its entire batch, amortising model loading cost.
3. **Chunk splitting**: Long segments (>250 characters) are split at sentence boundaries to avoid Chatterbox's 40-second audio output limit.

3.10.3 Quality Choice: Regular vs Turbo

The pipeline uses **Chatterbox Regular** rather than Chatterbox Turbo, despite Turbo being faster. Testing showed Regular produces 95% fewer TTS hallucinations (random word injection, phrase repetition, audio artifacts). For long-form content like podcast episodes, quality is worth the speed tradeoff.

3.10.4 Failure Handling

- Segments that fail TTS are tracked but don't abort the episode
- **20% failure threshold**: If more than 20% of segments fail, the entire episode is aborted to prevent short/broken output
- Failed segments produce silence gaps rather than corrupted audio

3.11 Stage 11: Audio Assembly

The final audio is assembled from pre-recorded show elements and generated content:

Order	Component
1	Intro jingle (pre-recorded music)
2	AI-generated disclaimer
3	"Here's Daniel's prompt!" announcement
4	Original user prompt audio
5	Whoosh transition sound
6	AI dialogue (Corn & Herman)
7	LLM credit announcement
8	TTS engine credit announcement
9	Outro jingle

Processing pipeline:
1. All components converted to consistent format (44.1 kHz, mono, 16-bit PCM)
2. Concatenated via ffmpeg
3. Single-pass **EBU R128 loudness normalisation** to **−16 LUFS** with **−1.5 dB true peak**
4. Encoded as MP3 at 96 kbps (transparent for speech, 50% smaller than 192k)

3.12 Stage 12: Publication & Distribution

The final stage publishes the episode across multiple systems:

1. **Cloudflare R2**: Audio file, cover art, transcript PDF, and waveform peaks uploaded
2. **Neon PostgreSQL**: Episode metadata, tags, category, embedding, and transcript inserted
3. **Vercel Deploy Hook**: Triggers a rebuild of the Astro static site (with retry logic, up to 3 attempts)
4. **Wasabi S3**: Full episode backed up to archival storage
5. **n8n Webhook**: Post-publication webhook triggers downstream syndication (Telegram, social media)

Quality gates before publication:
- **Duration check**: Episodes under 10 minutes are rejected (ffprobe validation, with file-size fallback)
- **Script length**: Minimum 2,000 words required before TTS
- **Segment count**: Minimum 10 dialogue segments required

4 Safety & Fault Tolerance

The pipeline is designed to be **resilient to partial failures**. Most stages fail open, and critical failures are caught and reported.

4.1 Fail-Open Architecture

Several pipeline stages are non-critical and degrade gracefully:

Stage	On Failure	Impact
Episode Planning	Continue without plan	Less structured script
Research Coordinator	Continue without research	Relies on model knowledge
Script Review (Pass 1)	Use original script	No fact-checking pass
Script Polish (Pass 2)	Use reviewed script	May have verbal tics
Cover Art	Use default cover image	Generic episode artwork
Waveform Peaks	Skip peaks	No waveform visualisation
Wasabi Backup	Skip archival	No off-site backup
Prompt Backup	Skip prompt archive	Prompt not archived

4.2 Quality Gates

Hard failures that prevent publication:

- **Script too short**: < 2,000 words (model returned truncated or refused response)
- **Too few segments**: < 10 dialogue segments (script didn't match expected format)
- **Episode too short**: < 10 minutes duration (TTS failure produced short audio)
- **TTS failure rate**: > 20% of segments failed (systemic TTS problem)
- **Audio download failure**: File < 1 KB or download timeout > 120s

4.3 Shrinkage Guards

Both editing passes include shrinkage guards to prevent the LLM from accidentally truncating the script:

- **Pass 1 (Review)**: Rejects output if more than 20% shorter than input
- **Pass 2 (Polish)**: Rejects output if more than 15% shorter than input

This was implemented after early testing showed that review agents sometimes returned drastically shortened "corrected" scripts.

4.4 Recovery Storage

If an episode passes all quality gates but fails during publication (R2 upload failure, database error), the complete episode is saved to a recovery folder in R2:

- All generated files (audio, cover art, script, metadata) are preserved
- Recovery script (`pipeline/scripts/recover_episodes.py`) can republish failed episodes
- Error notifications are sent via email with recovery path details

4.5 Zombie Job Prevention

A top-level `try/except` around the entire pipeline ensures that **all crashes result in the job being marked as failed** in the database. Before this was implemented, pre-publication crashes would leave jobs in "running" status indefinitely.

4.6 Notification System

- **Generation started**: Email sent when script generation begins (includes title)
- **Error notification**: Email sent on any failure (includes error details and recovery path)
- **Job status API**: Real-time progress via `/status/{job_id}` endpoint

5 Cost Analysis

The pipeline is designed for minimal per-episode cost. All compute runs on serverless infrastructure with no fixed costs beyond domain registration.

5.1 Per-Episode Cost Breakdown

Service	Cost	Notes
Modal TTS (2 × T4)	~$0.20	2 workers × 10 min × $0.59/hr
Modal Orchestrator (CPU)	~$0.01	15 min × $0.04/hr
Gemini API (script + edits)	~$0.05	Flash models, free-tier generous
Gemini API (transcription)	Minimal	Single multimodal call
Fal AI (cover art)	~$0.01	Flux Schnell, single image
Cloudflare R2	Free	Free egress, minimal storage
Neon PostgreSQL	Free	Within free-tier limits
Vercel	Free	Hobby plan sufficient
Total per Episode	**~$0.27–0.40**	Varies with episode length

5.2 GPU Pricing Reference

GPU	Per Second	Per Hour
T4 (current)	$0.000164	~$0.59
A10G	$0.000306	~$1.10
L4	$0.000222	~$0.80
A100 (40 GB)	$0.001012	~$3.64

The T4 was chosen as the cheapest GPU that can run Chatterbox Regular in acceptable time. Upgrading to A10G would roughly halve TTS time but double GPU cost.

5.3 Monthly Cost at Scale

At the current publication rate of approximately 5–10 episodes per week:

- **Weekly compute**: $1.50–4.00
- **Monthly compute**: $6–16
- **Annual compute**: $72–192

Modal’s Starter plan includes $30/month in free credits, which covers most months entirely.

6 Technology Stack

Category	Service	Role
LLM (Script)	Gemini 3 Flash Preview	Script generation, review pass (with Google Search grounding)
LLM (Utility)	Gemini 2.5 Flash	Transcription, planning, metadata, polish, tagging, embeddings
TTS	Chatterbox Regular	Voice-cloned speech synthesis (parallel T4 GPU workers)
Image Generation	Fal AI (Flux Schnell)	Cover art generation
Compute	Modal	Serverless GPU containers, job queuing, parallel workers
Object Storage	Cloudflare R2	Audio, images, transcripts, voice conditionals, show elements
Archival Storage	Wasabi S3	Long-term episode and prompt backup
Database	Neon PostgreSQL	Episode metadata, job tracking, tags, embeddings
Web Framework	Astro	Static site generator for podcast website
Web Hosting	Vercel	Static site hosting with deploy hooks
Admin CMS	Next.js	Episode management, storage cleanup, deploy triggers
Recorder	FastAPI + Vanilla JS	PWA for voice recording and upload
Audio Processing	FFmpeg	Format conversion, concatenation, loudness normalisation
CI/CD	GitHub Actions	Auto-deploy pipeline, recorder, and website on push to main
Notifications	Resend	Email notifications for generation status and errors
Syndication	n8n	Post-publication webhook for Telegram and social media

7 Lessons Learned

7.1 Chatterbox Regular vs Turbo

When the pipeline first adopted Chatterbox TTS, the Turbo variant was used for speed. Testing revealed that **Turbo produces significantly more hallucinations**: random word injection, phrase repetition, and audio artifacts. Switching to Chatterbox Regular eliminated approximately 95% of these issues. The occasional hallucination that still occurs with Regular is minor enough to go unnoticed in a 20–25 minute episode.

Takeaway: For long-form audio content, model quality matters more than speed. A hallucination-free 12-minute TTS pass is better than an 8-minute pass with artifacts scattered throughout.

7.2 Parallel Workers + Cached Conditionals

Two optimisations reduced TTS time from 36+ minutes to approximately 10 minutes:

1. **Pre-computed voice conditionals**: Processing voice samples on every segment added 5–10 seconds of overhead each. Pre-computing embeddings once and caching them in R2 eliminates this entirely.
2. **Parallel workers**: Instead of processing 80 segments sequentially on one GPU, distributing them across 2 workers (configurable up to 4) provides near-linear speedup. Each worker loads the model once and processes its entire batch.

Takeaway: For embarrassingly parallel workloads like segment-level TTS, the overhead of distributing work across workers is negligible compared to the speedup. Modal's `starmap` API makes this trivially easy.

7.3 The Two-Pass Editing System

The pipeline originally used a single verification agent (Perplexity Sonar via OpenRouter) to fact-check scripts. This caused a production failure when the agent returned a 169-word "corrected script" instead of the full 4,000-word script. The pipeline published this truncated output as an episode.

The replacement two-pass system was designed with specific safeguards:

- **Raw output only**: Both passes return the complete script as raw text, not wrapped in JSON (which was causing truncation)
- **Shrinkage guards**: Automatic rejection if the script shrinks too much
- **Fail-open**: Both passes return the original script on any error
- **Same model family**: Using Gemini for both generation and review avoids cross-model compatibility issues

Takeaway: When an LLM is editing another LLM's output, explicit length validation is essential. Agents will sometimes "summarise" instead of "edit" if not carefully constrained.

7.4 Episode Memory

The pipeline includes an episode memory system that provides context about recent episodes for cross-referencing. After experimentation, the context window was limited to **only the 3 most recent episodes**:

- More episodes led to excessive cross-references that felt forced
- The hosts now direct listeners to search the website for older episodes
- Semantic search finds contextually relevant past episodes (not just chronologically recent ones)

Takeaway: More context is not always better. A focused, relevant subset produces more natural references than a comprehensive history.

Appendix: Pipeline Stage Summary

No.	Stage	Model / Tool	Fail Mode
1	Audio Ingestion	HTTP + ffprobe	Hard fail
2	Transcription	Gemini 2.5 Flash	Hard fail
3	Research	Gemini 2.5 Flash	Fail-open
4	Episode Planning	Gemini 2.5 Flash	Fail-open
5	Script Generation	Gemini 3 Flash Preview	Hard fail
6	Review (Pass 1)	Gemini 3 Flash Preview + Grounding	Fail-open
7	Polish (Pass 2)	Gemini 2.5 Flash	Fail-open
8	Metadata	Gemini 2.5 Flash	Hard fail
9	Cover Art	Fal AI (Flux Schnell)	Fail-open
10	TTS	Chatterbox Regular (T4 GPU)	20% threshold
11	Audio Assembly	FFmpeg	Hard fail
12	Publication	R2 + Neon + Vercel	Recovery storage