> **TL;DR** AutoShorts.ai and Canva will get you to "good enough" in an afternoon. This post is for people who need something different: full control over TTS voice, custom Remotion compositions, deterministic artifact storage, and a publish layer you own end-to-end. The tradeoff is infrastructure you have to build and operate. Here is exactly what that looks like.

---

1 Why build your own pipeline instead of using a SaaS tool

The honest answer is that most of the time you should not build your own pipeline. Canva, AutoShorts.ai, OpusClip, and Pictory solve the 80% case: you give them a long video or a topic, they give you clips, you publish.

Build your own if you need one or more of these things:

- **Voice fidelity at the character level.** If you have a fine-tuned checkpoint for a specific speaker (e.g. FEMALE_01 from the IMDA NSC corpus) and you need that exact voice, no SaaS tool gives you that. You need to own the TTS call.
- **Composition programmability.** Remotion lets you express video structure as React components with TypeScript props. KaTeX equations, animated code blocks, data-driven charts - these are impossible in template-based editors.
- **Artifact ownership.** Your rendered video, the captions JSON, the TTS audio file, the QA report - they all need to be in your storage, addressable, versioned, and deletable on your schedule.
- **Multi-platform publish logic you control.** If you need the same render to hit LinkedIn, YouTube, TikTok, and Instagram with platform-specific caption formatting and retry handling, you need code you wrote.
- **Near-zero marginal render cost.** On a self-hosted GPU, a 60-second Short costs roughly the same whether you render it once or fifty times. That economics makes 136-render iteration cycles viable. SaaS pricing models punish that pattern.

If none of those apply to you, stop reading and open AutoShorts.ai. This guide is not for "faceless YouTube channel" automation - those tools optimise for volume

at minimal cost. This is for builders who need compositional control and iteration depth at production quality.

One important risk to acknowledge upfront: YouTube's July 2025 policy update renamed "repetitious content" to "inauthentic content" and explicitly targets mass-produced, template-based AI videos lacking originality. A custom pipeline that produces differentiated compositions is better positioned than a SaaS tool generating near-identical outputs - but you still need to understand the policy boundary. See the AI content rules guide for the full breakdown.

---

2 What 136 render cycles taught us about pipeline architecture

Across 38 Remotion sessions, one number stands out: 136 renders, 533 TTS cycles, 419 user turns, 34 calendar days - all for a single video composition.

That is not a pathological case. It is the gold standard for a complex composition (TTS + music + SFX + animated math). The iteration statistics by composition type:

Complexity tier	Renders	TTS cycles	Calendar time
Simple (1 composition, pre-recorded audio)	2–5	0	1 day
Medium (custom TTS, 5–10 composition steps)	10–20	20–60	3–5 days
Complex (TTS + music + SFX + B-roll)	50–136	100–533	2–5 weeks

The implication for architecture: every decision you make needs to be evaluated against the 136-render case, not the 5-render case. Authentication overhead, artifact versioning, billing gates, concurrency caps - they all compound over 136 cycles. A design that is "fine for testing" can become a GPU cost leak or a race condition in production.

Three specific lessons from those 38 sessions:

1. **Phases 2–6 form a tight loop, not a waterfall.** The first pass through scripting, TTS, composition, and render is a draft. You will re-enter phase 2 (scripting) after watching a render many times. Design the system to make that re-entry cheap.

2. **TTS text is the highest-churn artifact.** Script text changes on almost every iteration cycle. Caching TTS output is only useful if you hash the exact input text. A single word change should invalidate the cache.
3. **The human QA step is not optional.** Models cannot watch video. Temporal quality (pacing, visual rhythm, whether the equation appears at the right moment relative to narration) requires a human to watch the output. Budget this time.

---

3 The 7-phase workflow

Every session, regardless of complexity, moves through the same seven phases. The iteration count determines how many times phases 2–6 repeat.

```
Phase 1: Analysis          read MDX / script source, extract key claims
Phase 2: Scripting         refine narration text, adjust pacing, timing cues
Phase 3: TTS               generate audio from text
Phase 4: Composition       wire audio + visual into Remotion composition props
Phase 5: Rendering         `npx remotion render`   mp4 artifact
Phase 6: QA / Review       human watches rendered video
Phase 7: Iterate           return to phase 2, 3, or 4 based on what broke
```

Phases 2–6 account for roughly 80% of all turns across every session. Phase 1 happens once per video. Phase 7 is not a phase at all - it is the decision to loop again.

The practical consequence: your orchestration layer needs a fast path for partial re-renders. If only the TTS changed, you should not need to re-run the full composition pipeline. If only a text overlay changed, you should not need to regenerate TTS. The Inngest `job/render-requested` event exists exactly for this - it re-enters the pipeline at the right phase given what changed.

---

4 TTS integration: IndexTTS2 on RunPod serverless

The TTS layer is the most production-sensitive part of the pipeline. A bad TTS generation blocks every downstream phase.

We use IndexTTS2 deployed on RunPod serverless with a fine-tuned checkpoint: `FEMALE_01`, checkpoint `model_step14000.pth`, 7.2 GB. The fine-tuning corpus is the IMDA NSC dataset. Full benchmark and finetuning methodology: [IndexTTS2 Finetuning on IMDA NSC FEMALE_01].

The TTS extraction priority when assembling the text payload:

```
1. normalizedBriefJson.script          AI-generated brief (highest fidelity)
2. normalizedBriefJson.caption         fallback if script absent
3. jobInput.rawText                    user-provided raw text
4. job.inputSummary                    last resort (compressed, may lose nuance)
```

This priority chain matters because the brief generation and TTS generation are separate pipeline steps. A render-requested event may arrive before the brief is finalized. The fallback chain ensures TTS always has something to work with, but the logs should flag which source was used - degraded TTS quality is diagnosable from this field.

RunPod serverless gives cold-start latency of 15–30 seconds for the 7.2 GB checkpoint. For production throughput, keep a warm instance if you are generating more than 10 videos per day. The cost of a warm pod ($0.44/hr for an A40) is usually cheaper than cold-start latency on a busy day.

For a broader comparison of TTS model options including CosyVoice, F5-TTS, VoxCPM, and GLM-TTS, see: [Best Open-Source TTS Models for Production in 2026].

---

5 Inngest orchestration: the job render publish flow

The full job lifecycle runs through five Inngest event handlers:

```
job/created
    load job + resolve org membership
    verify billing (credits or active subscription)
    dispatch render payload to worker
    set job.status = "rendering"

render_completed (worker callback)
    validate HMAC signature
    create artifact rows in DB (video, caption, thumbnail, transcript, qa_report)
    set job.status = "draft_ready"
    notify user

job/render-requested (re-render from editor)
    same flow as job/created dispatch path
    concurrency cap: 5 active renders per org
    previous render artifacts preserved until new render succeeds
```

```
job/publish-requested
    load publish attempt row
    fetch video + caption artifacts (latest isCurrent versions)
    call provider API with platform-specific payload
    record providerPostId + providerPostUrl
    set publish_attempt.status = "published"

data/cleanup (scheduled, 30-day grace)
    soft-delete expired artifacts
    no hard R2 deletion without explicit org request
```

The key design decision: billing is verified at three points, not one. Job creation, render dispatch (re-renders also cost credits), and publish (some platforms charge per-post in future billing models). Checking only at creation means a user who re-renders 50 times can exhaust credits on compute without any gate catching it until the next job creation attempt.

---

6 Worker boundary design: why the render worker is external

The Remotion render process is not suitable for a serverless Next.js environment. A 60-second Short at 1080p takes 3–8 minutes of CPU/GPU time and produces a 50–200 MB artifact. Vercel function timeouts (max 300 seconds on Pro) and memory limits (3 GB) make this impossible to run inline.

The worker is a separate service - either RunPod serverless or a self-hosted GPU node - that accepts a render payload and returns results via a callback webhook:

Worker contract (discriminated union):

```
render_completed:
  jobId: string
  runId: string
  artifacts: Array<{
    type: "video" | "caption" | "thumbnail" | "cover" |
          "transcript" | "brief" | "qa_report" | "export_bundle"
    storageKey: string
    publicUrl: string
    sizeBytes: number
  }>

render_failed:
  jobId: string
  runId: string
```

```
  errorCode: string
  errorMessage: string

artifact_uploaded:
  jobId: string
  artifactId: string
  storageKey: string
  publicUrl: string
```

Authentication between the Next.js app and the worker uses HMAC with timing-safe comparison on the `X-Worker-Secret` header. A simple string equality check is not acceptable - timing attacks can leak the secret character by character against a predictable-latency endpoint.

The worker dispatches to RunPod primary and falls back to a self-hosted GPU if RunPod returns a cold-start timeout. The fallback is not a nice-to-have - production render jobs have SLA expectations and RunPod has occasional availability issues during peak GPU demand.

---

7 Concurrency and billing gates

Two things that will hurt you in production if you do not design for them upfront:

GPU cost abuse. A user who triggers re-renders in a loop (either maliciously or via a UI bug) can run up GPU costs faster than any per-seat pricing model anticipates. The concurrency cap of 5 active renders per org is enforced at the Inngest function level using `concurrencyKey: org.id`. When the 6th render is requested, Inngest queues it rather than dispatching - this prevents runaway parallelism without blocking the user entirely.

Credit accounting on re-renders. Every render dispatch should deduct credits, not just job creation. The `job/render-requested` handler must check balance before dispatching. If balance is zero, the handler should reject with a descriptive error that the UI can surface - not silently fail and leave the job in a stale state.

Billing gate at publish time. This is the gate most builders forget. If your business model involves per-publish charges or platform-specific quotas, the `job/publish-requested` handler needs to verify entitlements before calling any provider API. A publish that succeeds on LinkedIn but fails on YouTube because of a quota check mid-flight creates an inconsistent state that is hard to explain to users.

---

8 Multi-platform publishing: one render, eight destinations

The render produces a single MP4 artifact. The publish layer handles per-platform adaptation:

Platform	Status	Notes
LinkedIn	Fully implemented	Video post + caption
YouTube	Implemented	Shorts endpoint, title + description
TikTok	Implemented	Direct post API
Instagram	Implemented	Reels endpoint
X (Twitter)	Implemented	Media upload + tweet
Threads	Implemented	Meta Graph API
Facebook	Implemented	Page video post
RedNote ( )	Implemented	Notes post format
Lemon8	Implemented	Lifestyle post format

Each platform gets a separate `publish_attempt` row. A failed publish on one platform does not block others - the Inngest function fans out platform publishes independently and records per-platform outcomes.

Caption handling is worth calling out explicitly. The caption artifact from the render pipeline contains raw text. Before publishing, the platform handler reformats it: LinkedIn gets 3000-character limit with hashtag injection, TikTok gets 2200-character limit with different hashtag rules, YouTube gets the description field (no hard limit but algorithm-relevant length is 200–500 words). The reformatting logic lives in per-platform formatter functions, not in the render worker - the worker should not know about destination platform constraints.

---

9 Artifact versioning and the re-render loop

Every artifact type tracks a `version` integer and an `isCurrent` boolean. When a re-render completes:

1. Existing artifacts of the same type have `isCurrent` set to `false`
2. New artifacts are created with `isCurrent = true` and `version = previous + 1`

3. The old artifact rows are retained until the soft-delete grace period

This matters for two reasons. First, it makes rollback possible: if a re-render produces a worse result (it happens - TTS models are not deterministic), you can revert to a previous artifact version without triggering another render. Second, it makes the publish audit trail clean: you can see exactly which artifact version was published to which platform, which is important for compliance and for debugging post-publish discrepancies.

The practical implementation in Drizzle:

```
-- When inserting a new artifact, mark all previous versions of the same type as not current
UPDATE job_artifacts
SET is_current = false
WHERE job_id = $jobId AND artifact_type = $type;

-- Then insert the new one with is_current = true
INSERT INTO job_artifacts (job_id, run_id, artifact_type, storage_key, public_url, version, is_current)
VALUES ($jobId, $runId, $type, $storageKey, $publicUrl, $nextVersion, true);
```

Never hard-delete artifact rows on re-render. The cost of storage (a few MB per artifact per version) is negligible compared to the cost of losing the ability to investigate a publish failure after the fact.

---

10 What this costs to run

Honest numbers based on the production system as of Q1 2026:

Compute (RunPod A40, 48 GB VRAM):

- On-demand render: ~$0.39/hour, 3–8 min per Short = $0.02–$0.05 per render
- Warm pod (always-on): $0.44/hr = ~$316/month
- IndexTTS2 inference: included in the render worker, no separate cost

Storage (Cloudflare R2):

- $0.015/GB/month storage
- $0.36/million Class B operations (reads)
- A typical Short render produces ~150 MB of artifacts (video + captions + thumbnail)
- 100 renders/month ≈ 15 GB ≈ $0.23/month storage

Orchestration (Inngest):

- Free tier covers 50k function runs/month
- At 136 renders per complex composition (5–6 Inngest steps per render): ~800 function runs
- A moderate production load of 500 renders/month fits within the free tier

Total for 500 renders/month (warm pod + R2 + Inngest free tier):

- ~$320–380/month infrastructure cost
- Per-render cost: $0.64–0.76

At the SaaS pricing tier, 500 video exports from tools like Pictory or AutoShorts runs 99–299/month - but you get no voice customization, no Remotion programmability, and no artifact ownership. The self-hosted pipeline is 1–2x more expensive in raw cost but delivers capabilities those tools cannot.

---

11 When NOT to build your own

Being honest about the failure modes:

Do not build your own if you have less than 3 months of runway to invest in infrastructure. The initial build - worker service, Inngest handlers, artifact storage, publish adapters - takes 4–8 weeks of focused engineering time. The ongoing maintenance (RunPod API changes, platform API deprecations, certificate renewals, monitoring) is another 2–4 hours per week indefinitely.

Do not build your own if you are pre-product-market-fit. If you are still testing whether video content works for your audience at all, the iteration speed of a SaaS tool is more valuable than infrastructure control. Get to 50 videos published and watch your retention curves before you invest in a custom pipeline.

Do not build your own if you need 50+ languages. Custom TTS fine-tuning at scale across many languages is a research problem, not an engineering problem. Commercial TTS providers (ElevenLabs, Azure, OpenAI) have already solved multi-language at production quality. Use them.

Do not build your own if you cannot staff the QA phase. The human watch-the-video step is irreplaceable. If you are too resource-constrained to watch every output before it publishes, the automation provides no safety net. A

broken render will publish. Consider whether YouTube retention curve signals can serve as a post-publish quality signal instead of a pre-publish gate - but this is a different risk posture.

For an honest accounting of what a production-grade pipeline actually requires beyond just render infrastructure, see: What a Production-Grade AI Video Pipeline Actually Needs.

---

12 FAQ

Q: Can I swap Remotion for FFmpeg-based rendering?

Yes. The worker boundary is abstracted - the orchestration layer does not care how the worker produces the MP4. FFmpeg pipelines are simpler for linear compositions (clip + voiceover + captions). Remotion earns its complexity when you need programmatic composition: conditional elements, data-driven layouts, React components with state. If your Shorts are simple talking-head clips with captions, FFmpeg is the right tool.

Q: Does the pipeline support live captions (burned-in subtitles)?

Yes, but this is handled in the Remotion composition, not in a post-processing step. The caption artifact from TTS (word-level timestamps from WhisperX or equivalent) is passed as a prop to the composition, which renders captions as React components synchronized to the audio timeline. This gives you full styling control. The alternative - burning captions in post with FFmpeg - produces the same output but loses the ability to adjust caption styling without a full re-render.

Q: How do you handle TTS failures mid-pipeline?

The Inngest step that calls the TTS endpoint is wrapped with retry logic (3 attempts, exponential backoff). If all three fail, the job status moves to `tts_failed` and a separate alert fires. The job can be manually re-triggered from the same text without losing any prior work - the failed TTS step did not write any artifacts, so the artifact table is clean.

Q: What happens if a publish fails on one platform but succeeds on others?

Each publish attempt is independent. A LinkedIn success does not block a TikTok retry. Failed publish attempts are retried up to 3 times with exponential backoff.

After 3 failures, the attempt status is `failed` and the user is notified with the platform-specific error message. The video remains in `draft_ready` state - the user can manually re-trigger the publish for the failed platform.

Q: Do you use streaming TTS or batch?

Batch. Streaming TTS is useful for conversational interfaces where you need to start speaking before the full sentence is generated. For video production, you need the complete audio file before you can set composition timing. Streaming adds complexity without benefit in this context.

Q: What monitoring do you run on the pipeline?

Sentry for application errors (worker callbacks, Inngest step failures). Inngest's built-in dashboard for function run history and retry visibility. Custom alerts on job status transitions that take more than 15 minutes (likely a worker hang). R2 storage cost alerts via Cloudflare dashboard. No custom metrics infra - the Inngest dashboard plus Sentry covers the production failure modes we have actually encountered.