The Tollbooth Is Gone: Rebuilding My Publishing Pipeline with AI Agents

By Austen Tucker · 2026

For most of my writing life, the hard part was not finishing the work.

The hard part was everything after finishing.

Formatting. Scheduling. Uploading. Writing metadata. Creating social previews. Sending email. Checking that the post went live. Fixing the thing that did not go live. Remembering which dashboard owned which part of the process.

That administrative layer became a tollbooth between writing and being read.

This post is about how I rebuilt that layer as an agent-driven publishing pipeline: the architecture pattern, where MCP fits, and the failure modes that showed up once the demo became a real system. The writing stays mine. The clerk work moves into software.

Start with the job description

When I design a system, I start with a hiring question:

>

That framing cuts through implementation noise. It avoids starting with a tool and backing into a use case. It asks what work exists in the system.

For this project, I found four jobs.

Role 1: The serializer

The serializer takes a long manuscript (40,000 to 96,000 words) and turns it into scheduled episodes: markdown files with YAML front matter, one per installment, ready for bulk import.

The important constraint: it splits by narrative beat, not word count. A 1,000-word target helps, but a Friday installment should not end mid-scene because a counter hit four digits. If the clean break lands at 700 words, the episode runs short. If it lands at 1,800, it runs long.

That one constraint makes the output feel authored instead of chopped. It is also where agents surprised me most. I expected rough cuts and got usable episode boundaries on the first pass.

Role 2: The delivery worker

The delivery worker has a smaller job.

Every day, it checks the schedule. If a post should publish today, it publishes the post. It does not make editorial decisions. It does not rewrite copy. It does not improvise.

It answers one question:

>

If yes, ship it.

That matters because agent autonomy works best when the job has a narrow decision surface. The delivery worker should not reason about the content. It should operate the schedule.

Role 3: The archivist

The archivist handles assets.

For my stack, that means image uploads to Vercel Blob and references back into the CMS. The goal was to keep media handling out of the backend and avoid turning every post into a manual upload ritual.

The archivist owns:

- Uploading images
- Returning stable asset URLs
- Attaching images to posts
- Keeping the CMS record clean

It is not glamorous work. That is why it belongs in the pipeline.

Role 4: The SEO manager

The SEO manager fills out the metadata I always avoid until the last possible moment.

It generates:

- Meta descriptions
- Open Graph titles and descriptions
- Alt text
- Social copy
- Slugs
- Excerpts

This runs as part of the upload flow. By the time a post enters the queue, it already has the metadata it needs.

Again, the point is not to remove judgment from the system. The point is to move repetitive work into a repeatable process.

The stack

The pipeline currently uses:

- Next.js
- Payload CMS
- The Payload MCP server
- Vercel Blob
- ActiveCampaign
- Postmark
- Claude skills

Payload CMS became the center of the system because it is extensible, Next.js-native, and supports MCP.

That last piece matters most.

MCP turns CMS operations into callable tools. An agent does not merely generate a blog post and ask me to paste it somewhere. It can call a tool that creates a post, sets a slug, schedules publication, attaches metadata, and updates the CMS record.

That changes the architecture.

The core workflow

The happy path looks like this:

1. I finish or revise a manuscript.
2. The serializer splits it into episodes and writes markdown files with front matter.
3. The upload flow imports those files into Payload.
4. The SEO manager fills out metadata.
5. The archivist attaches assets through Vercel Blob.
6. The delivery worker publishes scheduled posts.
7. ActiveCampaign or Postmark handles email delivery, depending on the message type.

The goal is boring on purpose:

>

Tool calls, not vibes

The most important engineering lesson I learned: agents become useful when they operate tools instead of producing advice.

A prompt that says "please create a blog post" gives you text.

A tool call that says createPost gives you a CMS record.

That distinction determines whether your workflow is automation or theater.

For this pipeline, I wanted the agent to perform concrete actions:

- Create post
- Update post
- Schedule post
- Attach asset
- Suppress newsletter
- Generate metadata
- Confirm publication state

Each action has a boundary. Each action can fail. Each failure can produce a useful error.

That is the difference between an agent workflow and a magic trick.

Design: where human judgment went

The first frontend attempt was Cursor-assisted. It worked, but the design system got messy fast. I ran into CSS conflicts, Tailwind specificity issues, and layout decisions that accumulated like dust bunnies with opinions.

So I started over with a design-system-first approach.

Claude design chewed through that while I played with the website, testing for usability, dead ends, and "vibes."

The useful shift was not "AI made it pretty."

The useful shift was that I spent my energy on higher-value design decisions:

- Typography
- Spacing
- Reading flow
- Navigation structure
- Accessibility controls
- Text density
- Motion preferences
- Contrast modes

I spent real time on reader controls for font size, contrast, line height, and reduced motion because that is where my judgment mattered. I did not need to prove I could personally wrestle every breakpoint into submission.

The agents helped most when the design system already had rules. Without rules, they produced plausible mush. With rules, they produced components I could review, adjust, and ship.

Email delivery: Postmark and ActiveCampaign do different jobs

Disclosure: I work on ActiveCampaign. This section reflects my experience wiring both tools into my own publishing stack.

Postmark and ActiveCampaign solve different problems.

Postmark is transactional email infrastructure. ActiveCampaign is a marketing automation and list-management platform.

I needed both.

Postmark was straightforward. The API has a clean mental model: call an endpoint, send an email. That made time-to-first-email low. For transactional messages, that simplicity helps.

The tradeoff: Postmark does not own scheduling. If you want a message to go out on a timer, your application needs to trigger it.

ActiveCampaign has more surface area because it manages lists, campaigns, automations, segmentation, and analytics. That power comes with a larger data model and more setup time.

The RSS campaign path took more discovery than I expected. The feed configuration lived deeper in the email design flow than I initially looked. Once configured, the field mapping worked well, but finding the right place to configure it cost time.

The engineering takeaway is simple:

>

The API may be only one part of the integration. The product model is the real integration surface.

Idempotency matters

The sharpest failure mode came from scheduling an email against an RSS feed.

If a scheduled post publishes late, retries, or changes state unexpectedly, the email layer can send more than once unless the publishing side guards against it.

That means the pipeline needs idempotency.

For this system, I want each post to carry a durable send state:

- Not queued
- Queued
- Sent
- Suppressed
- Failed

A scheduled worker should check that state before triggering email. If the post already sent, it should not send again, even if the scheduler fires twice.

This is ordinary distributed-systems hygiene, but agent workflows make it harder to forget. They trip over every rock they can't see. So that's why we dogfood.

(I mean, that and they give me the software for free, so that's nice.)

What I would build first next time

If I were rebuilding my site from scratch, I would start with three things before touching the frontend.

1. A canonical post state machine

Every post should have clear lifecycle states: draft, imported, scheduled, published, email queued, email sent, failed, suppressed.

Do this early. Retrofitting state after the pipeline exists feels like teaching a raccoon tax law.

Model your data first.

2. A dry-run mode

Agents should be able to say what they would do before they do it.

For bulk imports, dry-run output should include:

- Number of posts created
- Publish dates
- Slugs
- Newsletter flags
- Missing assets
- Metadata warnings

This makes the workflow reviewable without turning it manual.

3. A replay-safe delivery job

The delivery worker should tolerate duplicate triggers. Assume cron jobs retry. Assume webhooks fire twice. Assume a human clicks the wrong button.

If replaying the job can cause damage, the job is not done.

What this says about agents

This project changed how I think about agent readiness.

I do not trust agents as free-floating autonomous coworkers. I do trust them more and more as constrained tool operators.

That distinction matters.

Agents are useful when:

- The job is clear
- The tools are explicit
- The workflow has boundaries
- The system can validate the result
- Failure states are visible
- Human judgment stays near product decisions

They are much less useful when:

- The task has no definition of done
- The tool surface is vague
- The agent must infer hidden business rules
- Failure looks like success
- Nobody reviews the output

The pattern that worked here was not "let AI handle publishing."

The pattern was:

>

That is where agents become practical.

The real win

This pipeline does not make me write.

That still has to come from me.

What it removes is the tollbooth after the writing.

For years, the administrative layer between finishing a piece and publishing it was enough to stop the whole machine. Not because any individual step was hard, but because the pile of small steps became a wall.

Now the wall is software.

I can hand the system a folder of stories and a schedule. It can serialize, import, annotate, schedule, and deliver them. I can review the parts that need judgment. The rest moves.

That is the promise of agents that feels real to me: not replacing the creative act, not replacing engineering judgment, but turning repeatable operational drag into callable workflow.

The first story has been going out this week: a cyberpunk found-family story about outcasts finding love and belonging in a hyper-capitalist hellscape. It will publish one installment at a time, while the pipeline underneath it does its quiet little paper-route job.

The room is open again.

The mailbox has a pulse.