Seed to Scheduled

A proof-of-concept AI content engine for food bloggers — one screen from a day's theme to a photorealistic photo, copy for multiple platforms, and a card on the board.

> **The build is public.** Code: github.com/architect4hire/brandforge. The part worth stealing — the committed, layered prompt library that produced the pipeline component — lives at `docs/SCRUB-PIPELINE-PROMPTS.md`.

At a glance

The concept. Prove that a solo food blogger — or a small brand running six content channels — can go from "it's Taco Tuesday on the Grill Master channel" to a finished, photorealistic photo, platform-specific copy for multiple networks, and a scheduled post, from a single screen, with AI doing the heavy lifting and a human keeping two decisions.

The build. A working PoC on .NET Aspire and Angular: a guided Content Pipeline that chains a content seed, a photorealistic image (Venice.ai), and social copy for seven platforms (Azure OpenAI), then files the result into a DAM library and a Kanban content board — end to end, in one flow.

The best practice. The ~750-line pipeline component wasn't vibe-coded. It was built from a committed, layered **SCRUB** prompt library — Context → Plan → Execute (seven layers) → Verify — that ships in the repo as an open, reusable artifact. The prompts are the plan, and the plan is public.

The treadmill nobody sees in the highlight reel

A food blogger's actual job, most days, isn't cooking. It's the treadmill after the cooking.

One dish becomes a shoot. The shoot becomes a photo you're happy with. The photo becomes a caption — and then it becomes *seven* captions, because Instagram wants hashtags and a hook, Pinterest wants a keyword-dense description, X wants it under 280 characters, the blog wants something with actual paragraphs, and TikTok, Threads, and Facebook each want their own register. Then it has to be filed somewhere you can find it again, tracked so you know what's gone out and what hasn't, and scheduled. Every day. Across whatever channels you run — comfort food, baking, weeknight quick bites, grilling, plant-based, global.

Most tools solve one slice of that and hand you back to the treadmill for the rest. A generator makes an image. A different tool writes a caption. A spreadsheet tracks what's posted. A

scheduler queues it. Between each one is a copy-paste, a re-context, a file you'll lose. The work isn't any single step — it's the *seams* between them.

The proof of concept was aimed squarely at the seams: **can one guided flow own the whole chain, from a day's theme to a post that's ready to schedule, without ever dropping the operator back into tool-switching?**

The concept, stated precisely

RecipeForge (the app; BrandForge is the platform underneath) is a food-blogger content engine with six channels — Cozy Kitchen, Baker's Bench, Quick Bites, Grill Master, Plant Forward, World Flavors — each with its own photography style, brand voice, and content "memory." Layered on top is a weekly rhythm most food audiences already know: Meatless Monday, Taco Tuesday, Wine Wednesday, and so on. Pick a channel and a day, and the system already knows a great deal about what good looks like.

The PoC's thesis is a single screen — the **Content Pipeline** — that walks an operator from that choice to a finished, filed, board-tracked asset:

Two things about that flow matter more than the AI in it.

First, there are exactly two human checkpoints, and they're deliberate. The operator reviews the *content seed* before OpenAI runs, and reviews the *photo prompt* before Venice.ai renders. Everywhere else, the machine moves on its own — social copy auto-fires the moment you pick a photo; the save flow files the asset and creates the board card without ceremony. The design question wasn't "how much can we automate," it was "where does a human's judgment actually change the outcome?" Two places. So: two checkpoints. Right-sizing the automation is an architectural decision too.

Second, the seams are gone by design. One screen, one flow, one source of truth. Seed to scheduled, no copy-paste, no lost files. The image the operator keeps is the only one that survives — every unchosen variant is discarded on selection, so the DAM stays a library, not a junk drawer.

What's underneath

It's a right-sized PoC, not a platform someone read about. One .NET Aspire solution: an ASP.NET Core API, an Angular single-page app, a SQL database, and Azure Blob storage for the photography — orchestrated so `dotnet run` brings the whole stack up with a dashboard for health, logs, and traces.

> graph TB UI["Angular 19/20 SPA
standalone + signals"] --> API["ASP.NET Core API
.NET Aspire"] API --> OAI["Azure OpenAI
gpt-4o · social copy"] API -->

VEN["Venice.ai
photorealistic images"] API --> BLOB[("Azure Blob
DAM photography")] API --> SQL[("Azure SQL
DAM · board · scheduling")]

The AI is split across two providers on purpose — Azure OpenAI for language (seven platform voices from one concept) and Venice.ai for photorealistic food photography — with a small, opinionated set of services composing prompts, holding per-channel "memory" as versioned markdown, and tracking every generated prompt so a good one can be reused instead of re-discovered.

No microservices. No message bus. No orchestration layer the workload doesn't have the problem for yet. It's a proof of concept whose job is to prove the *loop*, and the architecture is sized to exactly that — with a clear, honest path to production rather than a pretense of already being there.

How it was built — and why the prompts are the real deliverable

Here's the part I actually want people to take away.

The centerpiece of the PoC — the Content Pipeline component — is about 750 lines of Angular, and it is the kind of component where AI code generation usually goes quietly wrong. It has a nine-state wizard, two review checkpoints that must not be skipped, a variant lightbox, an auto-firing social step, and a save flow that touches three services. Ask a model to "build me a content pipeline wizard" in one shot and you'll get 750 plausible lines with a skipped checkpoint, a social call in the wrong place, and every unchosen image left on disk. It'll compile. It'll demo. It'll be wrong in ways you find later, in production, by accident. That failure mode has a name in the SCRUB framework — the **Plausibility Trap** — and it is the entire reason the prompt library exists.

So the component was built from a **committed, layered prompt library** — `docs/SCRUB-PIPELINE-PROMPTS.md` — that existed before the code did and shipped in the repo alongside it. It follows the SCRUB chain end to end:

- **Context.** Load the exact reference files and make the model summarize the patterns it must preserve — signals, `firstValueFrom` bridging, the real response shapes — before it writes a line.
- **Plan.** Propose the architecture in prose — the nine-step state machine, which signals are derived, where the two checkpoints sit — and confirm it before any TypeScript exists.
- **Execute, in seven layers.** Types and signals first. Then the template in vertical slices — config and step rail, the seed/compose steps, the variant grid and lightbox, the social and save bar. Then the orchestration methods, on top of a template that already exists. Then

helpers and styles. Each layer small enough to review in minutes; each one constraining the next until the plausible output and the correct output are nearly the same thing.

- **Verify.** A PASS/FAIL checklist derived from the restrictions, run against the finished code: *Does any transition skip a review checkpoint? Does the social step fire anywhere but after image selection? Does selection discard only the unchosen variants? Are all seven platforms rendered? Any* `*ngIf` *that should be* `@if`*? Any* `localStorage`*? Any* `.subscribe()` *where orchestration should await?* Every check maps back to a restriction, because Verify exists for the one scenario that beats everyone: the day you review a diff too fast.

The restrictions are tiered — CRITICAL / IMPORTANT / PREFERRED — and every CRITICAL one carries its reason, because the reason is what tells the model *why* the plausible default is wrong here. "Discard only the unchosen variants, not the chosen one" isn't style; it's the difference between a library and a junk drawer. "Never skip the review checkpoint" isn't a preference; it's the human judgment the whole design is built around.

That library is the best-practice artifact, and it's public on purpose. The code shows you *what* was built. The prompts show you *how to build it again* — deliberately, reviewably, without the Plausibility Trap. A prompt is a specification, and specifications deserve engineering.

> **Real Talk**
>
> The impressive-sounding version of this PoC has an agent that "just builds your whole content calendar." I built the boring version on purpose: a guided flow with two places a human has to look, and a prompt library anyone can read. The flashy version demos better and fails in the dark. The boring version is the one a real food blogger could actually trust with their brand — and the one whose build you can audit line by line, because the plan is committed next to the code.

What the PoC proved

- **The whole loop works from one screen.** Channel + day → seed → photorealistic image → seven platform-specific captions → a filed DAM asset and a tracked board card. The seams are gone.
- **Two checkpoints are enough.** Automating everything except the seed and the prompt keeps a human's judgment exactly where it changes the output — and nowhere it just slows things down.
- **The prompt library is reusable leverage.** The pipeline component's plan is committed, layered, and readable. The next component — and the next contributor — starts from a specification, not from archaeology.

- **The architecture is honest about being a PoC.** It's sized to prove the concept, with a documented path to production rather than a costume of being finished.

The path to production is written down, not hand-waved: authentication and authorization, resilience policies around the AI calls, durable blob storage for every generated image, the full seven-platform copy set wired into the pipeline, Buffer scheduling, and CI/CD with telemetry. A PoC that names its own next steps is a PoC you can actually decide about.

The practical version

If you're building an AI-assisted feature and you want it to survive contact with production:

- **Automate to the judgment line, not past it.** Find the one or two places a human's decision changes the outcome, put checkpoints there, and let the machine run everywhere else.
- **Kill the seams, not just the steps.** The value in a content tool isn't a better generator; it's removing the copy-paste between generation, storage, tracking, and scheduling.
- **Write the prompts before the code, and commit them.** A layered prompt library is a specification with a commit history. It makes review possible, keeps errors local, and tells the next person *why* the code looks the way it does.
- **Tier your restrictions and attach reasons to the critical ones.** The reason is what teaches the model why the plausible default is wrong for your domain.
- **End with Verify, and keep it.** A PASS/FAIL checklist derived from your restrictions is the thing that catches you the day you review a diff too fast.