"New Year, New You" Portfolio Challenge

Jan. 12, 2026

What follows is a **high-level build plan**, ordered deliberately so that each step constrains the next and prevents architectural drift.

This is not a task list — it's an **assembly sequence**.

Phase 0 — Fix the constraints (before writing code)

Outcome: Clear boundaries that prevent overbuilding.

We decide up front:

- one container
- one public service
- static-first
- one AI integration
- no user state
- no database

This ensures the system remains inspectable, fast, and contest-appropriate.

Only after this do we touch a framework.

Phase 1 — Curate the content (before choosing UI)

Outcome: Authoritative, frozen portfolio data.

We:

1. Select a **small set of projects** (5–7 max)
2. Write a short, structured summary for each:
    - problem
    - approach
    - tradeoffs
    - repo link
3. Store this as plain data:
    - `projects.json` or `projects.yaml`
    - Markdown files for project deep dives

At this point:

- no AI
- no frontend
- no deployment

This becomes the *single source of truth*.

Phase 2 — Build the static site shell

Outcome: A complete portfolio **without** AI.

We:

1. Choose a static framework (Astro / Next static)

2. Implement routes:
    - `/`
    - `/portfolio`
    - `/project/[slug]`
    - `/how-i-work`
    - `/about`
3. Render everything from the curated data

If the project stopped here, it would already be a valid portfolio.

This step proves:

- navigation
- layout
- readability
- performance

No dynamic behavior yet.

Phase 3 — Introduce the AI boundary

Outcome: A clean seam between content and AI.

We:

1. Add a tiny backend API:
    - one endpoint for “intro”

    - one endpoint for “ask”
2. Write **fixed system prompts**
3. Inject portfolio summaries as context
4. Return a single response per request

Critically:

- no conversation memory
- no user personalization
- no streaming
- no retries

This phase is about *containment*, not capability.

Phase 4 — Connect frontend to AI (minimally)

Outcome: AI appears as an assistant, not a feature takeover.

We:

- add a button on `/home` → calls intro endpoint
- add a small form on `/ask-the-portfolio` → calls ask endpoint
- display responses as plain text

No animations. No typing effects. No avatars.

If AI fails, the site still works.

Phase 5 — Containerize and deploy

Outcome: Live, reproducible submission.

We:

1. Write a single Dockerfile
2. Build locally
3. Deploy to **Google Cloud Run**
4. Confirm:
    - public access
    - fast cold start
    - predictable URL

This locks the runtime environment.

Phase 6 — Embed and submit

Outcome: Contest-ready artifact.

We:

1. Embed the Cloud Run app in the DEV post

- Add required label:

```
--labels dev-tutorial=devnewyear2026
```

2.
3. Link GitHub repos clearly
4. State exactly what is original vs prior work

No last-minute features.

Assembly order (compressed view)

1. **Content first** (projects + writing)
2. **Static site** (fully functional without AI)
3. **AI API** (small, bounded, optional)
4. **UI hooks** (buttons and forms only)
5. **Container + Cloud Run**
6. **DEV embed + submission**

If you reverse any of these, complexity creeps in.

Why this order matters

- Content constrains AI
- Static pages constrain UX
- API constraints prevent “chatbot sprawl”
- Deployment last avoids wasted work

This is how you build something **simple that stays simple**.