ENGINEERING CASE STUDY

Architecture, Iterative Development, and the Case Against Autonomous Generation on an Unmoderated Public Surface

A Technical Report on the Design and Deployment of a Static Portfolio Website with an Embedded Conversational Assistant

Jeremy Dowdy

M.S. Candidate, Western Governors University

July 2026

This report documents engineering decisions, build history, and architectural reasoning as of the date above. Where specific software versions or model identifiers are referenced, they are explicitly time-anchored to reflect what was current at the time of this project's development, not a claim of continued currency.

Abstract

This report documents the design, iterative construction, and architectural decision-making behind a personal portfolio website and its embedded conversational assistant. The system was built as a single-page, dependency-free static site hosted on GitHub Pages, and its development is presented here as a structured engineering case study rather than a promotional overview. Two architectures are described in full: a *production architecture*, which is fully static and carries zero backend dependencies, zero external API calls, and no server-side attack surface; and a *reference architecture*, which was designed, implemented, and validated end-to-end but deliberately withheld from deployment. The reference architecture bridges the static frontend to a hosted large language model (Anthropic's Claude API) through a Cloudflare Worker acting as a secure edge proxy, and its construction surfaced three distinct engineering failure modes – unhandled request methods, cross-origin resource sharing preflight failures, and credential propagation errors – each resolved and documented as general lessons for similar integrations.

The intellectual core of the report is Section 4, which presents a structured decision analysis of whether to deploy the fully autonomous, LLM-backed assistant on a public-facing, unmoderated resume site. The analysis concludes that although the autonomous pipeline was built and functioned correctly, the risk-to-benefit ratio of deploying an unguarded generative system on a professional portfolio did not justify the engineering investment required to properly moderate it. The production system instead uses a deterministic, client-side, keyword-matched rule engine, selecting from a fixed, pre-approved set of responses with zero network calls and zero possibility of off-brand generation. A companion case study (Section 5) documents a substring-matching defect discovered in that rule engine's early implementation, presented in the style of a formal engineering postmortem, including root cause analysis and resolution via word-boundary matching.

The report closes with a summary of current production status, a roadmap describing the conditions under which the validated LLM-backed architecture would be appropriate for a future, higher-stakes deployment, and an appendix containing the full reference implementation of the Cloudflare Worker proxy. Collectively, the report treats engineering restraint – the decision of *where not* to apply available capability – as a documented and defensible design outcome in its own right.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Personal portfolio websites are typically treated as design artifacts: a matter of layout, color, and copywriting. This report takes a different view. It treats the construction of one such site – a single-page profile for an IT professional and U.S. Marine Corps veteran, together with an embedded conversational assistant – as a small but complete systems-engineering problem, worth documenting with the same rigor normally reserved for production software: explicit architecture, a build log of what changed and why, a formal evaluation of a major design alternative that was ultimately rejected, and a postmortem of a defect that reached a working system before being caught.

The site itself is unremarkable in scope: it is a static, single-page application with no build pipeline and no backend. What makes the underlying engineering worth examining is not the site's complexity but the reasoning that shaped it, particularly around a single, non-obvious question. A working, end-to-end integration with a hosted large language model (Anthropic's Claude API) was designed, implemented, and validated during development. It was never deployed to the live site. That decision – to build a capability fully and then withhold it – is unusual enough to warrant its own structured treatment, and it forms the analytical center of this report (Section 4).

The remainder of this report proceeds as follows. Section 2 presents both the deployed production architecture and the validated-but-unused reference architecture, each illustrated with a system diagram. Section 3 traces the frontend's iterative development across three build cycles, documenting problems encountered and the reasoning behind their resolution. Section 4 is the analytical core of the report: it presents the reference LLM integration in full, evaluates Claude as the candidate intelligence layer, and works through a structured risk/benefit analysis that concludes with the decision to ship a deterministic assistant instead of an autonomous one. Section 5 presents a focused postmortem of a substring-matching defect discovered in the deterministic rule engine. Section 6 summarizes the system's current production status, and Section 7 outlines the conditions under which the shelved architecture would become appropriate for a future deployment. Appendix A reproduces the full reference implementation of the Cloudflare Worker proxy as a numbered code listing, and the References section that closes the report lists the external

technical documentation this work draws on.

Throughout, the report favors a plain and falsifiable style of claim: every architectural detail, defect, and design decision described here traces directly to the underlying build history, and no metric, date, or technical outcome is presented that was not part of that history.

Chapter 2

System Architecture

Two distinct architectures are described in this section. The first (Section 2.1) is the system that is actually deployed and publicly reachable. The second (Section 2.2) is a fully engineered alternative that was built and validated but intentionally never shipped. Presenting both side by side is deliberate: the production system can only be properly understood in contrast to the capability that was available and chosen against, a decision analyzed in full in Section 4.

2.1 Production Architecture (Deployed)

The live system is intentionally minimal. It is a fully static, single-page application with zero backend dependencies, zero external API calls, and zero server-side attack surface. Table 2.1 summarizes its two functional layers.

Table 2.1: Production architecture: layers, technology, and role.

Layer	Technology	Role
Presentation	`index.html` with embedded CSS and embedded JavaScript	The entire application. Hosted via GitHub Pages [6].
Assistant Logic	Client-side JavaScript (deterministic rule engine)	Answers visitor questions using whole-word keyword matching against a curated response set. No network calls.

Critically, there is no proxy layer and no third-party inference API anywhere in the production build. This is a deliberate simplification rather than an oversight, a distinction that matters because a system with an absent capability reads very differently from a system that deliberately declines to use an available one. The full rationale for that decision is developed in Section 4.3. Figure 2.1 illustrates the deployed topology: a visitor's browser communicates with a single static origin, and the assistant logic resolves entirely within that browser's JavaScript runtime.

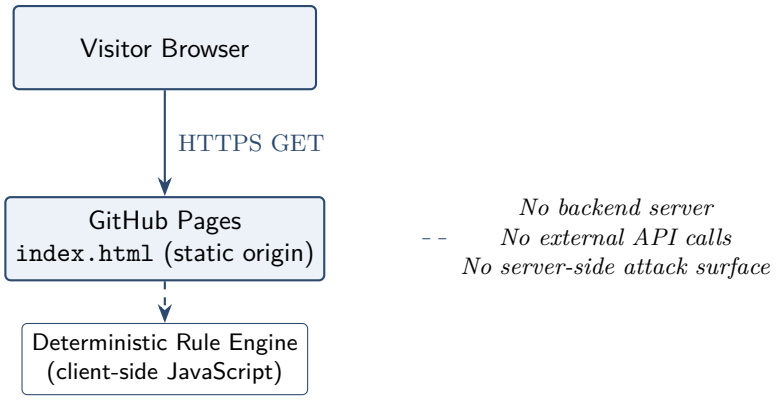


Figure 2.1: Production architecture: a single static origin serving all presentation and assistant logic. The rule engine executes entirely client-side with no outbound network calls.

2.2 Reference Architecture (Prototyped, Not Deployed)

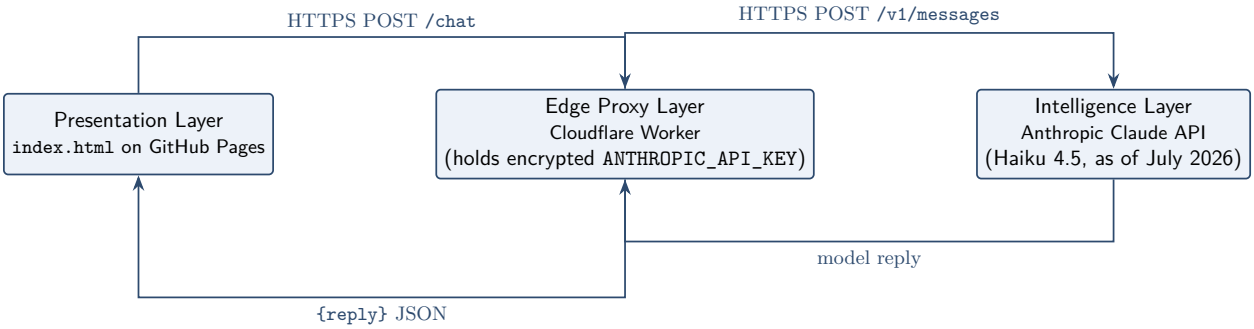
A second architecture was designed, built, and validated end-to-end during development, but was intentionally withheld from production. It follows the standard three-tier pattern for safely bridging a static frontend to a hosted large language model:

1. **Presentation Layer** – the same static `index.html`, sending user messages to a serverless endpoint instead of a local function.
2. **Edge Proxy Layer** – a Cloudflare Worker [1] acting as a secure intermediary. This is the layer that makes LLM integration on a static site possible at all: it holds the API key server-side, in an encrypted secret store, never in client-side JavaScript, and forwards requests to the model provider.
3. **Intelligence Layer** – Anthropic's Claude API, specifically the Haiku 4.5 model[1], selected for its low per-token cost and strong performance on constrained, single-purpose conversational tasks.

This architecture is fully documented in Section 4 and reproduced in full in Appendix A as a reference implementation: validated, functionally correct, and ready to deploy should the calculus described in Section 4.3 change for a future project. Figure 2.2 illustrates the request path across all three tiers.

The two architectures share a presentation layer but diverge entirely below it. Section 4 evaluates the reference architecture on its technical merits before explaining why, despite functioning correctly, it was not promoted to production.

[1]Claude Haiku 4.5 was the fastest and least expensive current-generation Claude model available at the time of this project's development, July 2026. Any reader encountering this report after subsequent model releases should read this and all later references to Haiku 4.5 as a historical statement of what was current at the time, not a claim of continued relevance.



Client secret never present in browser code; CORS headers enforced on every response path (Section 4.1.2)

Figure 2.2: Reference architecture: a three-tier proxy pattern bridging a static frontend to a hosted LLM. The Cloudflare Worker isolates the API credential from client-side code and mediates every request.

Chapter 3

Iterative Build Log: Frontend and Structural Evolution

The production frontend was not designed in a single pass. It evolved across three identifiable iterations, each triggered by a concrete problem encountered during development rather than by a planned feature roadmap. This section documents each iteration as a discrete unit: objective, implementation, and, where applicable, problem and resolution.

3.1 Iteration 1 – Establishing the Foundation

The initial objective was to build a high-performance single-page profile highlighting an IT professional and U.S. Marine Corps veteran background, without a framework or build step. The implementation is a single self-contained HTML file incorporating a responsive viewport, CSS backdrop filters (`backdrop-filter: blur`) for a modern glassmorphism aesthetic, and an absolute background image layer. No external dependencies, no build tooling, and no package manager are involved; the entire site ships as one file. This foundational constraint – a single static file with no toolchain – is what later made the zero-backend production architecture described in Section 2.1 the natural default rather than an added restriction.

3.2 Iteration 2 – Asset Management and Logo Rendering

Problem. Early attempts to render official tech-stack logos via external public image URLs (for example, logo SVGs hosted on third-party sites) triggered hotlinking blocks and cross-origin resource restrictions. Browsers rendered broken asset placeholders instead of the intended logos.

Resolution. Rather than fight cross-origin policy on assets outside the site's control, the implementation pivoted to a clean, dependency-free, text-styled footer. This guaranteed visual consistency across every browser and eliminated an entire category of environment-dependent failures – the familiar case of an asset that renders correctly in local development but breaks once served

from a different origin in production.

This iteration is a small instance of a pattern that recurs at larger scale later in the report: when an external dependency introduces a class of failure that is difficult to fully control, replacing it with a self-contained alternative can be the more robust engineering choice, even at some cost to visual fidelity. The same tradeoff – controllability versus capability – resurfaces in far more consequential form in the decision analysis of Section 4.3.

3.3 Iteration 3 – AI-Assisted Development Disclosure and Trademark Boundaries

As the build progressed with the help of an AI coding assistant (Claude, via Claude.ai), a decision was made to disclose that assistance directly on the site rather than let a recruiter assume otherwise. Two constraints shaped the implementation:

- **Accuracy.** The credit needed to describe the assistant's actual role – helping *build* the site – without implying that the *live chatbot* was itself calling an LLM at runtime. These are two distinct claims, and conflating them would have been misleading once the production architecture was finalized as fully static (Section 2.1).
- **Trademark boundaries.** Anthropic's trademark guidelines require pre-approval to use their logo and specify that the approved image asset is supplied directly by Anthropic rather than freely available for reuse [5]. Rather than risk unauthorized use of a brand mark, the credit was implemented as a text-only badge, styled to match the site's existing translucent pill-button aesthetic, linking out to Anthropic's site instead of reproducing their logo.

The result is a small footer badge reading "Development assisted by Claude," visually consistent with the rest of the page and factually precise about what it is claiming. This attention to precise, load-bearing wording in a footer credit is a minor detail in isolation, but it reflects the same discipline that governs the much larger claim addressed in Section 4: distinguishing what a system *could* do from what it is actually built and authorized to do at runtime.

Chapter 4

Embedded Assistant: Architecture Evaluation and Design Decision

The central engineering question for the assistant component was not whether an LLM-backed chatbot could be built – a working prototype was built and validated end-to-end – but whether it *should* be deployed on this particular site. This section documents both paths in full: the reference implementation that was built (Section 4.1), the evaluation of Claude as its intelligence layer (Section 4.2), and the structured decision analysis that determined the outcome (Section 4.3). This section is the analytical core of the report; the remaining sections describe what was shipped, but this section explains *why*.

4.1 Reference Implementation: Bridging a Static Site to a Hosted LLM

To connect a client-side chat widget safely to a hosted language model without exposing API credentials, a Cloudflare Worker was written to sit between the frontend and the model provider's API, implementing the Edge Proxy Layer of Figure 2.2. Three engineering problems surfaced during this build. Each is documented below as a general lesson, independent of the specific model provider involved.

4.1.1 Hiccup 1: The Direct-Hit Crash

Visiting the Worker's URL directly in a browser triggers a `GET` request. Because the Worker was written to expect a structured JSON payload delivered via `POST`, an unhandled `GET` request produced an unhandled parsing exception rather than a graceful error.

The fix was explicit method checking, performed before any attempt to parse the request body, shown in Listing 4.1.

```
1 if (request.method !== "POST") {
2   return new Response("Method␣not␣allowed", { status: 405 });
```

```
}
```

Listing 4.1: Explicit HTTP method validation, applied before request-body parsing.

4.1.2 Hiccup 2: CORS Preflight Failures

Browser security policy blocked requests originating from the GitHub Pages domain to the `*.workers.dev` endpoint, surfacing as opaque "Load Failed" errors in the browser console with no further diagnostic detail. This is a common failure mode when a static frontend and its serverless backend are served from different origins, and is documented in general terms by the Cross-Origin Resource Sharing specification [7].

The fix was explicit `OPTIONS` preflight handling and CORS response headers attached to *every* response path, including error responses, as shown in Listing 4.2. Omitting CORS headers from error paths is a common variant of this bug: a request that fails for an unrelated reason can still surface as a CORS error in the browser console, obscuring the real cause.

```
"Access-Control-Allow-Origin": "https://<your-github-pages-domain>",
"Access-Control-Allow-Methods": "POST, OPTIONS",
"Access-Control-Allow-Headers": "Content-Type"
```

Listing 4.2: CORS headers required on every response path, including error responses.

4.1.3 Hiccup 3: Secret Propagation

After the bridge was technically functional, requests to the model provider still failed. Root cause analysis identified two compounding issues: the API key had initially been stored as a plaintext environment variable rather than an encrypted secret, and, separately, environment variable changes in Cloudflare Workers require an explicit redeploy to propagate to the edge network – saving the value alone is not sufficient [2].

The fix was to store the credential using Wrangler's secret management (`wrangler secret put`), which encrypts it at rest and injects it into the Worker's runtime environment without ever exposing it in source control or client-side code, followed by an explicit `wrangler deploy` [3].

Taken together, these three hiccups characterize a recurring theme in edge-proxy integrations: correctness failures cluster at the boundaries of the system – unexpected request shapes, cross-origin trust, and credential lifecycle – rather than in the core logic of the proxy itself.

4.2 Evaluating Claude as the Intelligence Layer

With the proxy pattern validated, Anthropic's Claude API was selected as the model provider for the fully autonomous version of the assistant, for three reasons.

1. **No enterprise gate.** Access requires only a standard API key from the Claude Console and pay-as-you-go billing, with no enterprise sales process required to begin development [4].

2. **Cost profile.** Claude Haiku 4.5 – the fastest and least expensive current-generation Claude model at the time of development, July 2026 (see also the anchoring note in Section 2.2) – is well suited to a narrow, single-purpose conversational task of this kind. At its published rate, realistic traffic for a portfolio site runs to cents or low dollars per month.

3. **Scoping via system prompt.** Rather than requiring a vector database or retrieval pipeline, the entirety of the site's About, Projects, and Certifications content is small enough to embed directly in the system prompt, with explicit instructions to answer only from that content and redirect anything else.

A working end-to-end version of this pipeline – Worker, secret management, and scoped system prompt – was built and tested successfully. The reference pattern is included in Appendix A. It is important to be precise about what this establishes: the autonomous assistant was not abandoned because it failed to work. It worked. The decision that follows in Section 4.3 is a decision made *despite*, not *because of*, a successful technical result.

4.3 The Decision: Deterministic Rules Over Autonomous Generation

Despite having a working LLM integration, it was not deployed to production. This section presents the reasoning as a structured decision analysis: the deployment context, the risk profile of each candidate architecture, and the resulting judgment.

4.3.1 Deployment Context

A resume and portfolio site is a public-facing, unmoderated surface. Any visitor can type anything into the chat input, with no account creation, no rate-limited access, and no human moderation in the loop. This context matters more than any property of the model itself: the same underlying LLM integration could be entirely appropriate behind a different set of controls, and entirely inappropriate here. The evaluation that follows is therefore a statement about *fit to context*, not a general verdict on LLM-backed chat interfaces.

4.3.2 Risk Profile of an Unguarded Autonomous Assistant

An LLM-backed assistant, by design, generates a plausible response to *any* input unless it has been deliberately constrained with content moderation, abuse detection, rate limiting, and conversational guardrails layered on top of the base model call. A system prompt instructing the model to

stay on topic is a request, not an enforcement mechanism, and a request is not a control. Building an actual enforcement layer – moderation, abuse detection, rate limiting, guardrails – is substantive engineering work in its own right, comparable in scope to the proxy layer itself. Doing that work *incompletely* is arguably worse than not doing it at all: a half-guarded assistant on a professional resume site is a single bad-faith visitor away from an embarrassing, screenshot-able failure. Figure 4.1 depicts the additional control layer that a production-grade autonomous deployment would require and that this project deliberately did not build.

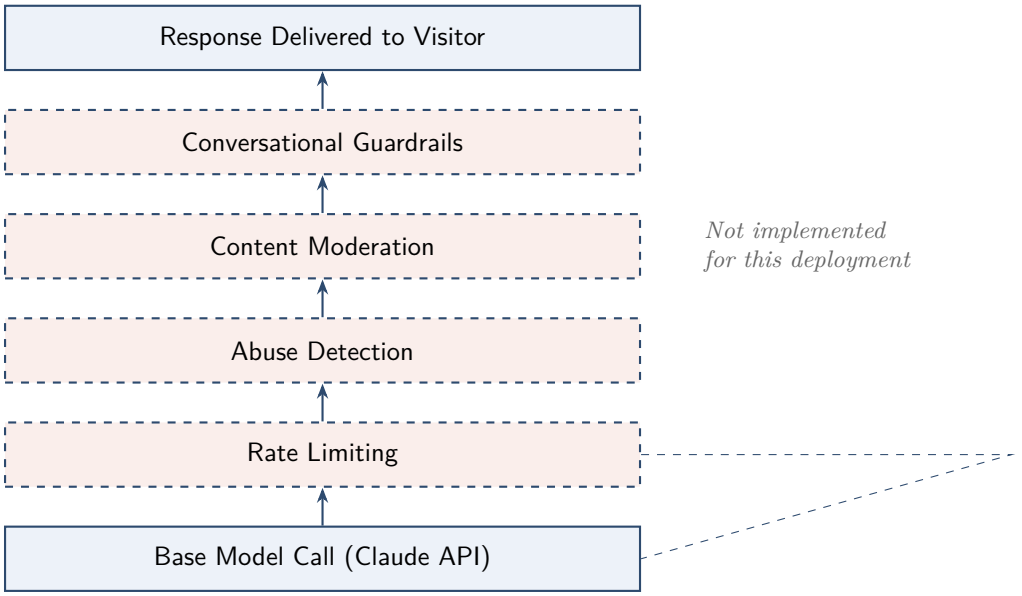


Figure 4.1: The control layers separating a raw model call from a response suitable for an unmoderated public-facing surface. The base model call and final delivery (solid) were validated; the four intermediate control layers (dashed) were identified as necessary but not built, which is the basis of the decision in Section 4.3.

4.3.3 Comparative Analysis

Table 4.1 summarizes the tradeoff between the two candidate architectures along the dimensions relevant to this deployment context.

4.3.4 Judgment

For a personal portfolio site, the risk-to-benefit ratio summarized in Table 4.1 does not clear the bar for deployment. The upside of a fully generative assistant here is marginal: it answers a fixed, small set of questions about one person's background. The downside of an unguarded generative assistant is disproportionate for a surface that exists specifically to make a good first impression on recruiters and hiring managers, where a single bad transcript can outweigh months of otherwise solid engineering work.

Table 4.1: Decision analysis: autonomous LLM assistant versus deterministic rule engine, for an unmoderated public-facing portfolio site.

Dimension	Autonomous LLM Assistant	Deterministic Rule Engine
Response scope	Unbounded; any input can elicit a generated reply	Fixed set of pre-approved responses plus fallback
Moderation required	Yes – content moderation, abuse detection, rate limiting, guardrails (Figure 4.1)	None – nothing is generated
Failure mode	Plausible-sounding but unintended or off-brand output	Fallback response; cannot go off-brand
Marginal upside for this site	Small – answers a fixed, small set of questions about one person's background	N/A (already achieves the same coverage)
Network dependency / cost	External API call per message; ongoing marginal cost	None; zero network calls, zero marginal cost
Engineering investment required to deploy safely	Substantial (full guardrail stack)	Already complete

That calculus changes for a production or client-facing product, where investing in a full moderation and guardrail stack is justified by the scope and stakes of the deployment. This decision is documented explicitly here for that reason: it reflects a judgment about *where* to spend engineering effort, not a gap in the ability to build the full version. The working prototype described in Section 4.2 and reproduced in Appendix A demonstrates the underlying capability directly.

The production solution instead is a deterministic, client-side, keyword-matched rule engine: roughly twenty curated response rules plus a fallback, all resolved with zero network calls, zero marginal cost, zero external dependency, and a guaranteed on-topic response in every case. It cannot be manipulated into generating anything off-brand because it is not generating anything – it is selecting from a fixed, pre-approved set of responses. This property, and a defect that nonetheless slipped past it, is the subject of Section 5.

Chapter 5

Case Study: A Substring-Matching Bug in the Rule Engine

Determinism eliminates one class of failure – unpredictable generated output – but it does not eliminate all classes of failure. This section presents, in postmortem format, a defect discovered in the keyword matcher underlying the deterministic rule engine described in Section 4.3.

5.1 Problem Statement

The initial keyword matcher used naive substring matching (`String.includes()`) to decide whether a rule applied to a given visitor message. This produced a false positive: the greeting rule, keyed on the word "hi," incorrectly fired on any message containing the word **his**, since "his" contains "hi" as a contiguous substring. A legitimate question such as *"What is his product management philosophy?"* was misrouted to a generic greeting response instead of the intended, substantively different answer.

5.2 Root Cause Analysis

Substring matching has no concept of word boundaries: it matches wherever a sequence of characters appears, including inside unrelated words. The matcher was, in effect, asking "does this character sequence occur anywhere in the message," when the actual intent was "does this word occur in the message." Those are two different questions, and the implementation answered the wrong one. The defect is a category of bug common to any system that treats natural-language input as an opaque character stream rather than as tokenized words: the check is syntactically simple and passes casual testing, because short, deliberately chosen test phrases rarely happen to contain a keyword as an accidental substring. The bug therefore did not surface during initial development and was only discovered under realistic phrasing.

5.3 Resolution

The matcher was rewritten to use word-boundary regular expressions (`\bkeyword\b`) rather than raw substring checks, ensuring each keyword matches only as a complete word or exact phrase. The corrected implementation is shown in Listing 5.1.

```
function getBotResponse(userText) {
  for (const rule of chatbotRules) {
    const matched = rule.keywords.some(kw => {
      const pattern = new RegExp('\\b' + escapeRegex(kw.trim()) + '\\b', 'i');
      return pattern.test(userText);
    });
    if (matched) return rule.response;
  }
  return fallbackResponse;
}
```

Listing 5.1: Word-boundary regular-expression matching, replacing naive substring matching.

5.4 Lessons Learned

This is a small bug with an outsized lesson. Even "simple" string-matching logic has edge cases that surface only under real usage patterns, and a deterministic system still requires the same rigor as a probabilistic one when it comes to input handling. The decision analysis in Section 4.3 argued that a rule engine eliminates the risk of *generating* an off-brand response; this case study is a reminder that it does not eliminate the risk of *selecting the wrong pre-approved response*, which is a narrower but still real failure surface that required its own verification.

Chapter 6

Current Status and Results

Table 6.1 summarizes the production state of each system component as of July 2026.

Table 6.1: Production status summary by component.

Component	Status
Portfolio website	Fully static, live, zero backend dependencies, zero external API calls, zero attack surface introduced by the assistant component.
Embedded assistant	Deterministic, rule-based system covering roughly twenty topic areas plus a graceful fallback for out-of-scope questions. Whole-word regex matching (Section 5) prevents substring false positives. A length-based simulated typing delay with random jitter and an animated typing indicator give the interaction a more natural, human-paced feel, without any of the unpredictability of a generative system.
AI-assisted development disclosure	A footer credit acknowledges Claude's role in the development process itself, implemented as a text-only badge to remain within Anthropic's trademark guidelines and to keep the claim precise: assistance during development, not autonomous operation at runtime (Section 3.3).
Reference LLM architecture	Validated end-to-end (Section 4.1, Appendix A); intentionally not deployed to production (Section 4.3).

Read together, these outcomes describe a system whose deployed footprint is deliberately smaller than its demonstrated engineering capability. That gap is not a shortfall; it is the documented output of the decision analysis in Section 4.3, and it is treated here as a result in its own right rather than as unfinished work.

Chapter 7

Future Work and Roadmap

The validated, working reference architecture for full Claude LLM integration described in Section 4.1 and reproduced in Appendix A remains available and is not treated as abandoned work. Its status is better described as staged: ready to deploy for a future project where the surrounding conditions justify it.

The specific condition identified in Section 4.3 is deployment context. The reference architecture is best suited to a production or client-facing system where the additional investment in moderation, abuse detection, rate limiting, and conversational guardrails is proportionate to the scope and stakes of the deployment – for example, a product with a defined user base, an operational team able to monitor and respond to misuse, and a business case where the marginal value of generative, open-ended conversation clearly exceeds that of a fixed response set. Under those conditions, the same Cloudflare Worker proxy pattern, secret-management approach, and system-prompt scoping strategy validated in this project would transfer directly, with the guardrail layer of Figure 4.1 built out as an explicit engineering deliverable rather than deferred.

In the interim, the deterministic rule engine remains the correct architectural choice for this specific site, and future iteration on the assistant component is expected to focus on expanding its curated rule set and refining matching precision, following the same discipline established by the fix documented in Section 5, rather than on migrating to autonomous generation.

Appendix A

Appendix: Reference Implementation

This appendix reproduces, in full, the reference Cloudflare Worker implementation described in Section 4.1 and evaluated in Section 4.2. It is included for completeness as documentation of the validated-but-not-deployed reference architecture of Figure 2.2, and reflects the Claude API's Messages endpoint conventions as of July 2026 [4]. It is reproduced verbatim from the project's build history; the model identifier embedded in the request body (`claude-haiku-4-5-20251001`) refers to the same Haiku 4.5 model discussed in Section 2.2 and is likewise anchored to July 2026, and should be read as a historical record of the specific model version used during development rather than a current recommendation.

The API key is never present in this listing: it is injected at runtime by Cloudflare's encrypted secret store, provisioned out-of-band via `wrangler secret put ANTHROPIC_API_KEY` as discussed in Section 4.1.3.

```
// Cloudflare Worker: secure proxy between a static frontend and the Claude API.
// The API key lives only in Cloudflare's encrypted secret store, set via:
//   wrangler secret put ANTHROPIC_API_KEY

const SYSTEM_PROMPT = `You are a scoped assistant for [site owner]'s portfolio site.
Only answer using the site content provided below. Redirect anything else
back to what you can help with.

--- SITE CONTENT ---
[site content embedded here]
--- END SITE CONTENT ---`;

const ALLOWED_ORIGIN = "https://<your-github-pages-domain>";

export default {
  async fetch(request, env) {
    if (request.method === "OPTIONS") {
      return new Response(null, { headers: corsHeaders() });
```

```
    }
    if (request.method !== "POST") {
      return new Response("Method not allowed", { status: 405, headers: corsHeaders()
          });
    }

    const { message, history } = await request.json();
    const messages = [...(history || []), { role: "user", content: message }];

    const res = await fetch("https://api.anthropic.com/v1/messages", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        "x-api-key": env.ANTHROPIC_API_KEY,
        "anthropic-version": "2023-06-01",
      },
      body: JSON.stringify({
        model: "claude-haiku-4-5-20251001",
        max_tokens: 512,
        system: SYSTEM_PROMPT,
        messages,
      }),
    });

    const data = await res.json();
    const reply = data.content?.find(b => b.type === "text")?.text ?? "";

    return new Response(JSON.stringify({ reply }), {
      headers: { "Content-Type": "application/json", ...corsHeaders() },
    });
  },
};

function corsHeaders() {
  return {
    "Access-Control-Allow-Origin": ALLOWED_ORIGIN,
    "Access-Control-Allow-Methods": "POST, OPTIONS",
    "Access-Control-Allow-Headers": "Content-Type",
  };
}
```

Listing A.1: Reference Cloudflare Worker: secure proxy between a static frontend and the Claude API. Status: validated in a local/staging test; intentionally not deployed to production (Section 4.3).

References

[1] Cloudflare, Inc. *Cloudflare Workers Documentation*. https://developers.cloudflare.com/workers/. Accessed July 2026.

[2] Cloudflare, Inc. *Secrets – Cloudflare Workers Docs*. https://developers.cloudflare.com/workers/configuration/secrets/. Accessed July 2026.

[3] Cloudflare, Inc. *Commands – Wrangler, Cloudflare Workers Docs*. https://developers.cloudflare.com/workers/wrangler/commands/. Accessed July 2026.

[4] Anthropic. *Claude Platform Documentation: API Overview and Messages API*. https://platform.claude.com/docs/en/api/overview. Accessed July 2026.

[5] Anthropic. *Anthropic Trademark Guidelines*. https://www.anthropic.com/legal/trademark-guidelines. Accessed July 2026.

[6] GitHub, Inc. *GitHub Pages Documentation*. https://docs.github.com/en/pages. Accessed July 2026.

[7] Mozilla Developer Network. *Cross-Origin Resource Sharing (CORS)*. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Accessed July 2026.