ChatGPT

Architecture and Deployment Options for the Viewer/Editor Project

For a Next.js-based *Viewer* and *Editor* application (currently single-user for book-building), there are several key decisions: how to structure the codebase for a shared design system, where to host the apps, and how to manage the content. Below, we discuss the pros and cons of each approach in these areas, following a mobile-first focus for the Viewer and a desktop-oriented Editor.

Code Structure: Monorepo vs. Separate Repositories

To maintain a **shared design system** and reduce drift between the Viewer and Editor UIs, you can either organize everything in a **monorepo** or keep separate repositories with a shared library.

- **Monorepo (Single Repository for Editor, Viewer, and Design System):** In a monorepo, all apps and libraries live in one repository (e.g. the Viewer app, Editor app, and a shared UI components/design system package) [1]. This makes it easy to co-version changes across projects.
- *Pros:* Code sharing is straightforward – common UI components or utilities reside in one place and can be updated for all apps in a single commit [2]. The development experience is consistent (all projects use the same dependencies, configurations, etc.), and there's a single source of truth for the design system, making it harder for versions to drift out of sync [2]. In scenarios with multiple applications and shared code, monorepos are often recommended as they simplify maintenance and versioning [3].

- *Cons:* The repository can grow large and build processes become more complex as everything is in one project [4]. Without proper tooling, even small changes might trigger rebuilding the whole repo (though modern tools like Nx or Turborepo use caching to mitigate this). Additionally, if granular access control or open-sourcing one part is needed in the future, a single repo could complicate permissions [5] (not a concern for a single-developer project). Overall, the monorepo approach may involve some upfront setup complexity, but once in place it “makes versioning and maintenance easier” at the cost of a larger repo size [1].

- **Multiple Repositories (Separate Projects with a Shared Library):** This approach keeps the Viewer, Editor, and possibly the design system in independent repos (or packages). For example, the design system could live in its own repo and be versioned as an npm package that the Viewer and Editor consume [6].

- *Pros:* Each application can be developed and deployed independently on its own schedule. There is greater isolation – one app can be updated without affecting the other, and each repo has its own CI/CD pipeline [7]. This independence can be useful if the Viewer and Editor need to be released separately or if different team members focus on each [7].
- *Cons:* Sharing a design system across repos introduces overhead. Updates to shared components require publishing a new package version and updating each app, which can lead to **dependency**

drift (apps temporarily on different versions of the design system) [8]. Coordinating cross-cutting changes becomes slower – e.g. a style overhaul means multiple pull requests across repos and careful version management [9]. During development, you'd need to use tools like npm/yarn link or a private package registry to test changes in the design system, which is less convenient than having everything in one codebase [10]. In short, a multi-repo setup makes it easier to accidentally "fork or drift in library versions" and requires more effort to keep everything in sync [11]. There's also duplicate configuration work (linting, testing setup, etc. in each project) unless you manually keep them aligned [12].

Summary: Given that you have two closely related apps (Viewer and Editor) and want a unified design system, a monorepo is likely the more efficient route. It allows you to manage shared code in one place and apply changes across both apps easily [2] [13]. A multi-repo could still work, but the overhead of maintaining a separate design system package and the risk of version drift are significant downsides for a single-developer project.

Hosting Platforms: GitHub Pages vs. Netlify vs. Vercel

Currently, you are using GitHub Pages for hosting, but you're open to switching to a modern platform like Vercel or Netlify. Below is a comparison of these options:

- **GitHub Pages:** GitHub Pages is a free static hosting service tightly integrated with GitHub repositories (you deploy by pushing to a specific branch).
- *Pros:* It's completely free for public repositories and very simple if you're already using GitHub – just push your static build and GitHub Pages will serve it [14]. Custom domains and HTTPS are supported, and it's great for small personal projects or portfolio sites [15]. The setup is straightforward for static content: no external CI needed beyond GitHub itself.

- *Cons:* GitHub Pages **only supports static content** – there is no support for server-side rendering or backend functions [16]. This means certain Next.js features (like API routes, server-side rendering, or image optimization) won't work unless you pre-build everything. Performance and build flexibility are more limited compared to dedicated platforms [16]. There are also usage limits: repositories have a recommended size limit (~1 GB) and bandwidth is soft-limited to 100 GB per month, with a limit of 10 builds per hour [17]. Importantly, GitHub Pages **is not intended for commercial or high-traffic production use** – GitHub's terms state it's not to be used as a free hosting service for running a business or SaaS product [18]. If your project grows (multiple books, higher traffic) or needs dynamic features, GitHub Pages would become a bottleneck. In fact, experts note that GitHub Pages "clearly loses the fight" in capability — even for hobby projects you often get more by using Netlify or Vercel's free tiers [19].

- **Netlify:** Netlify is a popular cloud platform for hosting static sites and modern web apps (JAMstack style). It pulls from your Git repo and handles builds, deployments, and a global CDN automatically.

- *Pros:* Netlify offers a rich set of features beyond basic hosting. It has **continuous deployment** (auto-building your site on each git push), and supports **serverless functions**, built-in form handling, and an identity/auth service out-of-the-box [20] [21]. It provides deploy previews for pull requests, easy environment variable management, and other developer-friendly tools. Netlify's CDN is fast and tuned for static content, and the platform makes it easy to manage custom domains and HTTPS. The

interface is very user-friendly, and setup is quick (often just a matter of linking your GitHub repo) [22]. These integrated services (forms, auth, A/B testing, etc.) can be a big plus if you need them [21].

- *Cons:* The Netlify free tier imposes some limits – notably **300 build minutes per month** on the free plan (and 100 GB bandwidth) [23]. For a personal project with infrequent updates this is usually sufficient, but if you build very often or have large builds, you might hit that limit. Netlify's advanced features (like its add-ons for forms or identity) have usage caps on the free tier (e.g. 100 form submissions, 1000 identity users) with additional costs if you need more [24]. While deploying standard static sites is extremely easy, using Next.js on Netlify historically required an adapter plugin – however, Netlify now supports Next.js 13+ fairly well (including server-side rendering) via its plugin, so this is a minor concern. There could be a slight *learning curve* to fully utilize Netlify's more advanced features (if you venture into custom functions, etc.) [25], but basic usage is straightforward. Overall, Netlify provides a lot of functionality, but you'll want to ensure your usage stays within the free tier bounds or be ready to upgrade for heavy use.

- **Vercel:** Vercel is the company behind Next.js, and their platform is specifically optimized for Next.js applications (though it supports others as well).

- *Pros:* **Deep Next.js integration** is Vercel's biggest advantage. Features like server-side rendering, **Incremental Static Regeneration**, Image Optimization, API routes, and the new App Router features all work seamlessly on Vercel, often with zero configuration [26]. Vercel auto-detects your Next.js project and builds/deploys it with appropriate settings. It also offers a global edge network and supports advanced capabilities like Edge Functions and middleware for ultra-fast dynamic responses. Deployment is very simple (via Git integration or the Vercel CLI) and the developer experience is excellent. On the free plan you get a generous allowance of **Build Minutes (6000/month)** and **Bandwidth (100 GB/month)**, which typically outmatches Netlify's free build time allotment [23]. The platform provides **analytics**, preview deployments for each git branch, and a marketplace of integrations (for databases, CMS, etc.) [27]. If your app grows, Vercel's scaling and performance are top-notch, and many consider it the default choice for Next.js projects due to how well it supports the framework [26].
- *Cons:* Vercel's free tier, while generous, **does have limits**. For instance, 100 GB bandwidth may become an issue if your book app gets very popular (though you can upgrade plans as needed) [28]. Also, Vercel's terms require that if a project is for a commercial venture (e.g. a business or SaaS), you should use a Pro plan – the hobby/free tier is mainly for personal or hobby sites [29]. In practice, many small projects start on the free plan and only upgrade when needed, but it's worth noting the expectation. Vercel doesn't include some of Netlify's extras like built-in form handling or identity; you'd typically add external services for those if required. Overall, there are very few downsides for using Vercel for a Next.js app, aside from the potential need to move to a paid plan if you exceed the free limits or need enterprise support.

Hosting choice summary: For your project (which may expand to multiple books but not massive scale initially), **Vercel** or **Netlify** will provide a much smoother experience than GitHub Pages. GitHub Pages is excellent for simple static sites but it lacks the flexibility and has strict limits that could hinder a growing, modern web app [19]. Vercel would likely be the easiest fit if you plan to leverage Next.js features (since it's essentially built for Next.js) and want minimal configuration hassles [26]. Netlify is also a great choice, especially if you value its integrated add-ons or have other frameworks to host; it can handle Next.js just fine, and offers a more all-in-one solution beyond just hosting [21]. Both Netlify and Vercel have Git-based workflows and support atomic deployments, previews, and rollbacks, which is great for a continuous

editing/deployment cycle. You might choose to deploy the Viewer on one of these platforms for better performance and capability (and potentially deploy the Editor app there too, if you ever want to access it remotely or share it). Both platforms have free tiers that should cover your needs initially, with the option to scale up if your traffic or build frequency increases [23] [30].

(Note: If offline access for readers becomes important later, both Netlify and Vercel can serve Progressive Web Apps or use service workers for caching. The hosting choice doesn't prevent adding offline support when you get to that stage.)

Content Workflow: Local JSON Files vs. Online CMS

Currently, the Editor generates JSON files that the Viewer uses to render book content. Right now this is a **local, file-based workflow** (single-user). You're considering whether to stick with this approach initially or move toward a more dynamic content management solution (like a headless CMS or a hosted database with an online editor interface). Here are the trade-offs:

- **Local Files & Static Generation:** This means the content (chapters, sections, images, etc.) is stored in local JSON/YAML/Markdown files and the Viewer app statically builds pages from those. The Editor app would manipulate these files (possibly on your machine) to create the book structure.
- *Pros:* This approach is very **simple and efficient** for a single-author scenario. You have full control of content in your repository, and changes can be tracked with version control (Git). Static site generation offers great performance – the viewer can pre-render all pages, so end-users get fast, cached content without database calls [31]. It's also highly secure and stable (no live database or server to maintain, fewer points of failure). Maintenance is straightforward for a developer: you can "set it and forget it" with fewer moving parts [32]. For a content-heavy site that doesn't change constantly, static files are ideal [33]. In short, static generation is **perfect for a book or documentation site** where content updates are infrequent and can be bundled into periodic site builds – it yields fast load times and simple infrastructure [31].

- *Cons:* The biggest drawback is **workflow speed and collaboration**. Because content updates require rebuilding and deploying the site, they are slower and less convenient compared to using a CMS. Non-technical collaborators (if you had any in the future) would find it hard to update content without involving a developer. As one source notes, static site generators "require developers for setup and have slower workflows for content updates," whereas CMS platforms allow more direct editing [34]. If you needed to update content frequently or on a schedule, the manual build/publish cycle could become a bottleneck [35]. Also, if you eventually have *many* books or very large content files, managing them as raw files could become cumbersome (though not impossible). There's no web interface for editing (in the current setup), so you must run the Editor locally. This is fine for now, but it means you can't easily edit from anywhere or on the fly without your development environment. In summary, the static file approach sacrifices some flexibility and real-time updating capability in exchange for simplicity and speed – a trade-off that is usually acceptable for small, developer-managed projects [36].

- **Headless CMS or Online Editor:** This approach would involve storing content in a database or headless CMS and building an interface (or using an existing CMS UI) to edit and manage books. The Viewer could then fetch content via APIs or at build time from this source. Essentially, the content management is **decoupled** from the site code.

- *Pros:* An online CMS (headless or a custom admin) would make content updates much more **convenient**, especially if multiple books or frequent changes are expected. Content editors (even if it's just you for now) could update text or images through a web interface and see those changes reflected without manually redeploying. If using Next.js, you could even use Incremental Static Regeneration or on-demand revalidation to update the live site shortly after content changes, providing a near-real-time update experience. A CMS is designed for managing content at scale: it can handle rich media, multiple content types, and it's better suited if you ever have additional collaborators or non-developers contributing. In general, headless CMS solutions shine for **dynamic or large-scale content scenarios** – they provide flexibility for frequent updates, and the content is not tied to the build process [37]. If your project grows to have many books or needs to deliver content to multiple platforms (web, mobile, etc.), a headless architecture would facilitate that by decoupling content from presentation [38].
- *Cons:* The trade-off for using a CMS or custom backend is **increased complexity**. Implementing a headless CMS involves setting up and maintaining additional infrastructure (a database, CMS software or service, and possibly user authentication for the editor) [38]. This is a heavy upfront effort for a single-user project. One source emphasizes that a headless CMS "introduces a level of technical complexity that simply doesn't exist" in more straightforward setups [38]. You would need to design content schemas, handle hosting (unless using a SaaS CMS), and possibly build the editing UI if you don't use an off-the-shelf solution. There are also costs to consider: many headless CMS platforms have paid plans, especially if you need private repositories or advanced features. For a one-person operation, setting up accounts and managing deployment of a CMS might be overkill when you could just edit a JSON file. Additionally, with a decoupled system, you have to ensure the Viewer and the content API stay in sync (though this is manageable). In summary, moving to an online CMS now would solve some workflow issues but at the expense of significant initial setup time and complexity – which might not "pay off" until you truly need multi-user editing or very frequent content changes.

Given the current circumstances (single author, relatively infrequent updates, and a focus on ease of development), **keeping the file-based/static approach for now is a sensible choice**. It plays to the strengths of static sites (speed, simplicity, low overhead) [31] and avoids premature complexity. As the Core DNA reference notes, for primarily static content with less frequent updates, a static site generator is ideal, whereas a headless CMS shows its value when you have a large or frequently updated site with complex content needs [34]. You can certainly "cross the bridge" to a more dynamic content management solution later if needed – for example, if you decide to allow co-authors or if updating content via Git becomes too slow for your workflow. At that point, you could consider integrating a headless CMS or building a lightweight online editor backend. Until then, the local JSON + static build method will likely serve you well with minimal headache.

Conclusion

In summary, to set yourself up for success with this project (and future multiple books):

- **Use a Monorepo with a Shared Design System:** This will eliminate UI drift between the Editor and Viewer by keeping all components in sync. Monorepos are specifically beneficial when you have "multiple related front-end apps that share a lot of code (e.g., a design system) and you want to manage them in one place" [13]. The single repository approach streamlines development and

maintenance across both tools, which outweighs the slight increase in repo complexity for a project of this scale.

- **Move off GitHub Pages to a Modern Host:** Take advantage of platforms like Vercel or Netlify for better performance and features. Both offer continuous deployment and CDN coverage, which will be important as your content grows. Vercel is a natural fit for Next.js (with full support for Next's features and an optimized infrastructure) [26], while Netlify provides excellent static hosting plus extras if you need them [21]. Either would remove the limitations you'd face on GitHub Pages and give you more freedom to implement things like serverless functions or the new View Transition API for slick page animations. On free tiers, you won't pay anything unless your usage grows significantly, and you'll avoid the usage caps and restrictions of GitHub Pages [17] [19].

- **Continue with Static Content Workflow (for now):** Given that you don't expect other collaborators in the near term and can tolerate doing deployments for updates, the static JSON + Next.js approach is perfectly fine. It's fast and simple, and an ideal solution for content-heavy sites that don't require constant updating [36]. As your project evolves to multiple books, you can still manage that with a static approach (for example, generating a static site that includes a gallery of books and their content). Only consider introducing a headless CMS or live editing platform when it's clear that the project's complexity demands it – such as if you find yourself needing more frequent content changes or you want to enable edits without any developer intervention. Adopting a headless CMS too early would introduce unnecessary complexity and maintenance overhead [38]. You can defer that decision until the benefits outweigh the costs.

By starting with this setup – a monorepo for code consistency, a robust hosting platform, and a simple content pipeline – you'll have a solid foundation that is both **low-headache now and adaptable later**. This foundation will also make a great portfolio piece for you, showcasing a modern, well-architected app. When features like the View Transition API (for smoothly animating between portrait/landscape views, etc.) come into play, you'll be in a good position: both Next.js and modern hosts support the latest web APIs, so implementing such enhancements should be straightforward. Likewise, responsive design for mobile-first Viewer and a desktop-friendly Editor will be easier to maintain with a shared design system in the monorepo. Overall, this approach emphasizes upfront effort where it has high payoff (code maintainability and deployment reliability) and postpones it where it's not yet needed (collaborative content management and offline support), which aligns well with your project goals.

---

[1] [3] [10] Should You Use a Monorepo?
https://www.pronextjs.dev/workshops/next-js-production-project-setup-and-infrastructure-fq4qc/should-you-use-a-monorepo-w6lx7

[2] [4] [5] [6] [7] [8] [9] [11] [12] [13] Monorepo vs Monolithic vs Multi-Repo: Choosing the Right Approach for Front-End Development. | by Imonhossain | Medium
https://medium.com/@imonhossain02_4840/monorepo-vs-monolithic-vs-multi-repo-choosing-the-right-approach-for-front-end-development-c7dae105bd4b

[14] [15] [16] [20] [22] [25] [28] Hosting a Static Website: Comparing GitHub Pages, Netlify, and Vercel - NamasteDev Blogs
https://namastedev.com/blog/hosting-a-static-website-comparing-github-pages-netlify-and-vercel/

17 18 19 21 23 24 26 27 29 30 Best NextJs Hosting Provider? Netlify Vs Vercel Vs GitHub Pages - Jon D Jones
https://www.jondjones.com/frontend/jamstack/best-nextjs-hosting-provider-netlify-vs-vercel-vs-github-pages/

31 32 33 34 35 36 37 Headless CMS vs Static Site Generator: Key Differences Explained | Core dna
https://www.coredna.com/blogs/headless-cms-vs-static-site-generator

38 Headless CMS Pros and Cons: Should Your Next Project Go Headless?
https://www.netguru.com/blog/headless-cms-pros-and-cons