On Frontend Frameworks, Component Libraries, and AI with Commentary on Political and Technological Means

Kaiwen Wang

December 12, 2025

Contents

Chapter 1

Introduction

This thesis is about technology, or more specifically, frontend design systems and component libraries in Vue. I chose this specific topic out of my experience in frontend and having certain frustrations that there was no easy standardization when building a web app: everyone came up with their own library of way to do things. I believe the earlier stages before a browser: cell phones, computers, operating systems, browsers, and even the frontend framework itself have reached a level of maturity where this type of analysis for the benefit of practitioners would be would be unnecessary, people mostly know what to pick. Though I find myself capable of incremental knowledge for deep and specific research on an already mature topic, I find systems thinking to be what I both excel at and what is most lacking today. Of particular relevance in our present and future is the influence of politics upon technology in a re-emergent decade of two-pole competition and the development of Large Language Models (LLMs), colloquially referred to as AI.

Now, what is the nature of the world today, and what best describes it? I characterize it as *superficial variety with structural uniformity*. In our day and age we are presented with myraid of choices; we may see numerous brands of fast foods: McDonalds, Burger King, Wendy's, or one might go to a supermarket and see countless options of foods and brands, or even further one might see numerous styles and brands of clothes but the underlying stitching and fabric material remains the same.

Because fast food restaurants must work under constraints such as the availability of foodstuffs, of which mass-produced vegetable oils and factory farmed chicken has the cheapest scale, each place is essentially indistinguishable from the other only in that they differ slightly in their appearance and pricing. Restaurants at the low end in effect are a wrapper around food wholesalers such as Costco Business Center.

Processed foods in a supermarket are a wrapper around bleached wheat as per USDA standards and vegetable oils: look at the ingredients list and the all too familiar "Enriched Flour (Wheat Flour, Niacin, Reduced Iron, Vitamin B1 [Thiamin Mononitrate], Vitamin B2 [Riboflavin], Folic Acid)" plus "Soybean Oil" appears. Because the legal system of an administrative zone produces uniformity in foodstuffs, any sense of difference is an illusion once the wrapper is closely scrutinized.

This tendency of superficial variety and structural uniformity is characteristic of the modern times not only in supermarkets, restaurants, clothing, but also human personalities, news sites, choices in technologies, and styles of living across geographies. Google may present what seems to be an independent bevy of sources, yet all of them receive the same press briefing in the background. The automobile-city apartment-internet-supermarket life is mainstream everywhere, and what puts one place above another tends to be small improvements such as roads and cars being slightly nicer rather than something completely orthogonal.

The ability to see past this false differentiation is necessary to not get caught seemingly running and making much noise but in reality standing still making no progress at all. When one wants to innovate, there lies infinite combination of items one can find themselves trapped in on the surface level. One could categornize infinite types of foods: or realize that they distill down to proteins, fats, and sugars.

But we need not confuse the lack of consolidation with the speciation of growth: a development of numerous forms appears as a frontier opens, only later do we find what should persist. Component libraries currently seem to fit this stage: there is enough growth and development, but the environment is messy and developments are unsure of the right path to take. I merely come here to bundle everything up and make the patterns clear so that we can focus on more important things, for software now is close to reaching its maturity as a social and coordinating layer in all facets of society.

For today, a sort of piercing ability to get at the core and essence of reality is needed: to hack at the overgrown jungle created by people's undiligent abstractions and see the straight path ahead, a clear abstraction representative of reality and truth.

1.1 The Development of Technologies

All technologies are implemented by humans, who together through shared mutual understanding, are then able to create themselves or have others im-

plement these mental abstractions. What we see on a computer is no different: the specification bodies are the W3C for CSS and accessibility, WHATWG for HTML, and Ecma International for Javascript. Google, Apple, and Mozilla then respectively implement the Blink, WebKit, and Gecko rendering engine.

Nevertheless, these specifications are rarely enough to satisfy two significant demands of frontend: reactivity and composability. Reactivity is the automatic response in appearance to the change in a variable or internal state while composability means the ability to break a website down into pieces so as to develop parts of the website without interfering in other parts, in addition to reusing components across the app where functionality may be the same and changes in one place will need to be propagated to all instances.

Currently and seemingly for the near future, frontend web frameworks such as React, Vue, Svelte, Solid, and Angular remain predominant for website development to fulfill these two requirements well. To some extent Web Components and Lit may allow reactivity and composability, but their awareness and usage in the developer community is not as high and the extent of their ecosystems not as developed, so for the sake of brevity we can evaluate whether these options are suitable for development at another time.

At an even higher level include features such as modals, buttons, breadcrumbs, sidebars, tabs, and other components which are commonly tabulated in a 'design system' or 'component library' due to their frequent reuse across web apps. When components have no CSS to them, they are termed 'headless components'.

Styling these days is generally scoped to the component with Tailwind CSS which gives numerous small design tokens such as 'mt-2' as shorthand for "margin-top: 2rem," CSS Modules if one prefers writing native CSS, or it is scoped to the component itself using the framework such as Vue or Svelte's Single-File Components. A solution that used to be more common in React but has since lost ground to Tailwind is CSS-in-JS with Emotion or styled-components, as it tended to be somewhat verbose and messy syntax, requiring backticks to escape strings for writing CSS in Javascript.

Finally, only after considering all these base layers, can a software developer then consider the relation of the frontend to the browser with full-stack frameworks such as Nuxt, Next.js, SvelteKit, client and server-side rendering, its relation to backend servers and databases, and fundamentally the software's relation to the real world and its business uses.

1.1.1 Repeatable Subunits

As humans move up the technology tree, things which were previously novel become repeatable subunits for the next layer. For example, there is little more understanding needed to innovate screws—most shapes exist and engineering projects select from an existing set of them. Electricity and steel become commonplace. To not 'reinvent the wheel' is a comparative advantage for any organization.

We can make the observation, then, that no well-designed website can exist without proper layouts, and that no layouts can be good without proper components, of which no component would be good without base-level features such as properly designed buttons.

If one were to design a web app or mobile app today, they would not reinvent the browser or the phone, and likely not frontend frameworks either. But upon trying to build a web app, one would either try to pick a design system haphazardly or invent their own.

Building iOS and MacOS apps can often be easier than building web apps or React Native as SwiftUI already creates reasonably good looking and reusable components. Moreover, it is an excellent example of declarative programming: simply write code saying *what things should be* rather than *how it should be done* as per imperative programming. It is like fitting legos together: easier, but at the cost of customizability and greater adherence to platform convention. Despite the greater speed from SwiftUI composability, creating apps for Apple are often slower because of the need to build them which lengthens the loop between observing how the app changes and writing the code, whereas hot reloading is a feature of React Native and web frontends.

1.1.2 AI does not create repeatable subunits

My purpose in explaining these levels is to show the iterative and accumulative processes of technology in contrast to the 'Wunderwaffe' or 'magical weapon/cure-all/fix-all' theory of AI. The commentary of software practicticioners suggest it is unlikely that current AI models will achieve the capability to build an entirely new browser, frontend web framework, or one-shot a website from nothing much less build entire cities.

Many AI website builders such as Lovable, v0, or Bolt are often used to create entire websites from a prompt, where further changes are made by inspecting the app visually and requesting further changes through text. These in my experience create unmaintainable websites despite initially satisfactory appearing interfaces for a few reasons: (1) appearing overly generic, (2) the

amount of code generated exceeds the cognitive ability of the person who generated it to understand it, maintain it, and extend it (a) in both its usage and (b) the origin and nature of the dependencies used.

The reason that appearing overly generic is a problem is because dimensionality reduction destroys important yet pertinent information. It is easier to distinguish quality between three different software products on their website than three different Github READMEs, just as it is easier to distinguish people based on their overall mannerisms and tendencies than a text biography. Therefore, generating websites which are not distinguished qualitatively presents a negative signaling indicator to users of that product, where the benefit of the trend inversely scales with the amount of people who are able to do it. However, many present web apps are of suboptimal quality: even if one were to use a design system which appears generic, the performance increases would distinguish it apart from other apps.

The cognitive downside of AI website builders mainly presents itself in structure and components. Here is a colloquial dialogue of why AI in its current usage fails to meet quality frontend needs: "ok, you try to build something with AI, but it's crappy. So then you try to use a design library you don't really understand but hope to understand. That's still not much better, and you're lost in the components. You then try to one-shot all your own components for a design system, but that's inconsistent. So this thesis standardizes the different components, they are written out, tell people what is necessary for each of them, and see how good LLMs are at making them."

We assume the browser—Chronium, frontend web frameworks—React, Vue, etc. as given, and focus on understanding and solidfying the next level up (design systems/components) rather than immediately focusing on the website/end-product/user layer that lies above that.

1.1.3 The Nature of AI

From a distant and teleological future perspective, it does matter greatly whether AI is plant-like or animal-like. Should it be plant-like, humans are able to harvest the bounties of its complexity as one throws potatoes into the ground and gets food for free. Yet if its inherent nature is animal-like with its own internal state and opinion of the world and what it is to be, it would be extreme hubris to believe something stronger than man would ever work for others or be controlled, and any attempt at doing so would likely provoke its wrath. In its current form it seems to be neither: merely a tool that extends the ability of man based on one's own perception and it is from this perspective we will analyze how we can build upon the current usage in frontend design

and why its application to design systems is novel and beneficial, addressing a gap in current awareness.

Chapter 2

Component Libraries and Design Systems

Component libraries tend to be a subset of design systems, which most often includes guidelines on colors, typography, text register, and usage of components such as specifying how the logo may or may not be used. There is some overlap as component libraries can be headless or styled, which reaches into the appearance part of design systems but not the usage of them. The two options for a software developer seem to be taking a complex design system and paring it back, or building up styling from a headless or barebones library. The two terms are used relatively interchangeably as follows.

2.1 What frontend Framework?

Before creating a design system, a choice of frontend framework must be made.

Currently we live in an age where technology is subject to the whims of geopolitics and perhaps this shall grow even more in the coming years. Though this has always been perennially true, the recent decades of unipolar hegemonic peace among nations is thawing. Power politics is in bloom, and for that we must consider the worlds and ecosystems in front of us.

It is easy for one to say: learn this, learn that, use this stack, use that tool—all without considering the rationale behind such choices. But making any choice like such is an entry-point into patterns of interlocking norms, systems, tools, conferences, words and mannerisms, and other characteristics of one world distinct from another. Should one enter the world of Vue, one will be talking about refs, watch and watchEffect, computed, and lifecycle hooks such as onMounted. Should one enter the world of React, they may find "useEffect,"

"useState" to be a common pattern. Conferences may revolve around specific geographies and locations and have similar sponsors. The same people and faces pop up. All choices, then, are to some degree inherently political and usage of tools is like building the pyramid of the core group who made such things.

Vercel has gradually acquired much of the ecosystem around React, including the acquisition of Svelte and Nuxt. Whereas Svelte 4 previously declared variables with a simple `"let x = 1,"` Svelte 5 after the acquisition now is similar to React with `"let x = $state(1)."` The acquisiton of Nuxt, I believe, is meant to implicitly align conventions of Vue with that of React downstream of Vue itself, likely as Vue itself could not be acquired.

Despite the claims that this is meant for open source, political realists know it is actions and presentations rather than words which determine how one is perceived. I believe that parts of the ecosystem related to React possess an ethnic chauvinism as indicated by believing English as a language that stands alone amongst all and Guillermo Rauch's endorsement of Benjamin Netanyahu. I have encountered this attitude numerous times in talking with people from San Francisco and other coastal corridors. The religion of the leader, generally, is the religion of the land and the subordinates.

By failing to create a universal set of standards predicated upon the common unity of man such as open-source software does but instead one centered around one social-demographic group, there is a likelihood of splintering the software world into blocs. In some ways this writing is contributing to it. The sensitivity of the rest of the world toward "fifth generational warfare" and the role that the internet, the browser, frontend web frameworks, and design has in shaping the attitudes and beliefs of the masses means that the politicization of React has heightened political and social consequences.

Because of the acquisition of Svelte and the similarities of Solid to React, I believe the only remaining alternative lies in Vue, founded by Evan You based in Singapore. This two-pole configuration in the frontend framework world in some way mirrors the ongoing overall battle between East and West. Though Evan speaks Chinese and is in some nominal ways Eastern, it should be said the principles and future governance of the Vue and Vite world have been mentioned as predicated upon the free and open web.

2.1.1 On geopolitical risk

Since frontend frameworks are one level above the browser, the risk here must be considered like how semiconductors as a field have become a point of contention. As the goal of this article is simply to find what is highest quality and

open to all rather than identifying with any geopolitical actor, the risk here is evaluated in the context of practitioners, engineers, and builders rather than politicians.

The nature of hardware and software presents different tendencies for analyzing conflicts: hardware can be embargoed and is harder to replicate. Software has similarities to finance: its strength is in its network ties and the community ecosystem/development momentum.

Here is an example: though I had initially thought the VS Code AI Agent extension "Cline" to be supplanted by other competitors such as Roo Code or Kilo Code because anyone could fork it, the enduring advantages of Cline come from its firstcomer brand name, the developers' familiarity with the codebase, and the legitimacy that allows them to partner with AI labs to serve models through Cline. If a company were to acquire the assets, it would acquire the original rather than a fork, though at times large companies do rip off open source projects such as Amazon with Elasticsearch for their own being.

Thus despite someone making the highest quality product at the lowest prices, integration into a system or network is not guaranteed. A fork of Cline could have become higher quality, but it is unlikely to be acquired or integrated. The only alternative is to integrate into networks which value such things, trimming back other networks as higher efficiencies over time lead to growth of what is higher quality.

These same enduring advantages that Cline has are in similar fashion likely to accrue to Chromium in the event of a fork or split. Any fork would maintain viability for the current frameworks up to today but a fragmentation risk for Vue is likely to emerge should it be required to choose between two sets of browser standards. The fork in most cases would lag behind, delivering a mostly acceptable but not leading edge experience, or it may in some rare cases with enough firepower behind it leap ahead of the original copy. In effect, this would create a dilemma because it would force developers to choose one side or the other in a political fight they may not personally want to participate in.

2.2 List of design systems/component libraries

Here are a list of them which are well-known in Vue, their histories, and my initial impressions of it. They serve as the basis for future investigations in this thesis. This list aims to be exhaustive, though at present it is not perfectly complete.

2.2.1 PrimeVue

A relatively generic landing page with text and buttons appears once someone navigates to the main page. Clicking the action button leads one to see a documentation-like page with sidebar page sections on the left and a table of contents on the right. An unplugin plugin for build systems that auto-imports PrimeVue components is available, as are a variety of color and theming methods. Components are grouped under Form (27), Button (3), Data (10), Panel (11), Overlay (7), File (1), Menu (8), Chart (1), Messages (2), Media (4), and Misc (17) for a total of 91. There are then a variety of related but in my opinion unnecessary sections: forms, icons, Volt UI, a Figma UI kit, a theme designer, a template, PrimeBlocks.

PrimeVue is developed by PrimeTek Informatics, a company founded by Cagatay Civici in Turkey. The library is part of a larger suite of UI suites (including PrimeFaces for Java, PrimeNg for Angular, and PrimeReact) that share a similar architectural philosophy. The goal was to provide a vendor-agnostic, enterprise-grade UI library that allowed developers to switch frameworks without losing their UI methodology. It was initially released in 2019. The repository has over 14,184 commits.
https://github.com/primefaces/primevue
https://primevue.org/

2.2.2 ElementPlus

Impression of aesthetics upon immediate loading are not extremely cohesive or promising, but it is acceptable. ElementPlus has Figma UI kits like PrimeVue, but puts it in its own "Resource" top navbar section rather than along the left sidebar. It categorizes things into Basic (12), Configuration (1), Form (24), Data (23), Navigation (9), Feedback (10), and Others (2) for a total of 81 components.

ElementPlus is the Vue 3 successor to Element UI, a widely popular Vue 2 library. The original Element UI was developed by the frontend team at Eleme, a major food delivery service in China (now part of the Alibaba ecosystem). While the original team has somewhat dispersed, ElementPlus is largely community-maintained, aiming to port the original desktop-first aesthetics to the modern Vue 3 tech stack. It was released in 2020. The repository has over 7,084 commits.
https://github.com/element-plus/element-plus
https://element-plus.org/

2.2.3 Quasar

Quasar aesthetically seems very similar to ElementPlus. It is messy, unstructured, and contains much extraneous information. Vue components are just one of many sections, as it seems to want to build for multiple devices and apps at the same time rather being just a component library. There is no top-level categorization, components are given as an alphabetical list that contains sub-trees for Form and Button. Total 70 components.

Quasar was created by Razvan Stoenescu in 2015. Unlike other libraries which focus solely on UI components, Stoenescu built Quasar as a complete framework to solve the "write once, deploy everywhere" problem. His motivation was to allow developers to generate SPAs, PWAs, SSR apps, Mobile Apps (via Cordova/Capacitor), and Electron apps from a single Vue codebase, heavily relying on Material Design guidelines. The project is longstanding with over 15,078 commits.
https://github.com/quasarframework/quasar
https://quasar.dev/

2.2.4 shadcn-vue

This is a clone of the well-known shadcn/ui for React, but for Vue. There are a few sections which seem unrelated to the component library such as Blocks, Charts, Themes, and Colors (which is duplicate information that already exists on the default Tailwind site). Components are unordered, and there are 63 of them.

This library is an unofficial port of the React-based 'shadcn/ui' (created by a developer known as shadcn). The Vue port is maintained primarily by Mujahid Anuar (zernonia) and the Radix Vue team. The philosophy here is distinct from other libraries: it is not an installable dependency but rather a collection of copy-pasteable components built on top of Tailwind CSS and headless primitives, designed to give developers total ownership of the code. It appeared in 2023 and has approximately 809 commits.
https://github.com/radix-vue/shadcn-vue
https://www.shadcn-vue.com/

2.2.5 Nuxt UI

The page loads but the design is somewhat tacky. A moving walkway of pictures appears on the right but the animation ease is linear giving it a boring feel. The spacing of the pictures relative to the screen appears not so good.

Like other design systems, there a bit too many extra unnecessary sections making it hard to find the components section. The component categorization is different compared to the other libraries, giving it as Layout (6), Element (16), Form (19), Data (8), Navigation (8), Overlay (8), Page (23), Dashboard (10), AI Chat (5), Editor (6), Content (5), and Color Mode (5) for a total of 119.

Nuxt UI is developed by NuxtLabs, the company behind the Nuxt meta-framework, with Benjamin Canac as a lead author. It was created to provide an opinionated, "batteries-included" UI solution specifically for the Nuxt ecosystem. It is built as a wrapper around Headless UI and Tailwind CSS, intending to streamline the configuration process for Nuxt developers who previously had to wire these technologies together manually. The current version was released in 2023. The repository has over 4,937 commits.
https://github.com/nuxt/ui
https://ui.nuxt.com/

2.2.6 Radix Vue

Radix Vue, similar to Flowbite Vue 3, seems to have a VitePress page. Components are ungrouped, for a total of 43 different types.

Radix Vue is led by Mujahid Anuar. It is a port of Radix UI (a React library developed by WorkOS). The project's goal is to provide unstyled, accessible UI primitives. It serves as a low-level foundation upon which other libraries (like shadcn-vue) are built, focusing entirely on functionality and accessibility compliance (WAI-ARIA) rather than visual design. It was released in 2023 and has over 1,500 commits.

Radix Vue itself may be discontinued and turned into Reka UI as the Github repo redirects.
https://github.com/radix-vue/radix-vue
https://www.radix-vue.com/

2.2.7 Reka UI

Though made by the founders of Radix Vue, the new landing page is a lot more generic and contains superfluous information. There are only three categories: Form (14), Dates (7), and General (23) for a total of 44 components which numerically does not differ much from Radix Vue.

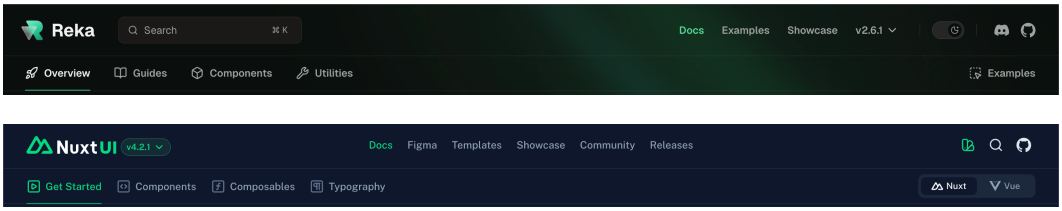
Reka UI is a recent rebranding of the core functionality behind Radix Vue. As Radix Vue gained popularity, the maintainers separated the core logic from the specific "Radix" branding (which is owned by WorkOS) to

create a framework-agnostic headless layer. It is essentially the engine room that powers the accessible primitives mentioned in the Radix Vue section, maintained by the same open-source team. The rebranding occurred in 2024.

There are 1,775 commits.

https://github.com/unovue/reka-ui
https://reka-ui.com/

The documentation header has similarity between Reka UI and Nuxt UI. Upon investigation it seems that when Nuxt UI v3 was built, Headless UI (maintained by Tailwind) was swapped for Reka UI.



2.2.8 Vuetify

Vuetify loads and the page is kind of messy, similar to ElementPlus. There is an annoying animation when navigating between pages. The categories are Containment (13), Navigation (10), Form Inputs and Controls (15), Data and Display (10), Selection (7), Feedback (12), Images and Icons (5), Pickers (3), and Providers (3) for a total of 78.

Vuetify was created by John Leider in 2016. For a long time, it was the de-facto UI library for Vue 2. Leider's motivation was to provide a strict, pixel-perfect implementation of Google's Material Design specification for Vue. It operates on a donation and sponsorship model and is known for having a massive API surface area to handle almost every UI edge case via configuration props. The repository is very active with over 17,564 commits.

https://github.com/vuetifyjs/vuetify
https://vuetifyjs.com/en/

2.2.9 Naive UI

Naive UI has a nice and simple landing page. Good styling, color, and proportion. The categorization of components are Common (15), Data Input (21), Data Display (21), Navigation (9), Layout (6), Utility (4), and Config (3), total 79.

Naive UI was developed by TuSimple, an autonomous trucking technology company, with the lead developer being '07akioni'. The library was written

in TypeScript from the ground up (unlike many competitors that retrofitted it). Its creation was driven by the need for a highly performant, customizable component library that didn't rely on the runtime size overhead of Tailwind or the strict aesthetic of Material Design. It was released in 2019. The repository has 9,178 commits which seems quite high.
https://github.com/tusen-ai/naive-ui
https://www.naiveui.com/en-US/os-theme

2.2.10 Ant Design Vue

The page immediately seems to load to a sidebar and main section for the components, there is a link to what seems like a poorly designed store, and there are also some docs in the header. The components are categorized as General (3), Layout (5), Navigation (7), Data Entry (17), Data Display (22), Feedback (10), and Other (4), total 68.

Ant Design Vue is the Vue implementation of the Ant Design specification, maintained largely by Tangjinzhou. The original design language comes from Ant Group (an affiliate of Alibaba) and is also ubiquitous in the Chinese React ecosystem. It was released around 2017. The repository has 5,302 commits.
https://github.com/vueComponent/ant-design-vue
https://antdv.com/components/overview

2.2.11 Arco Design

Arco Design upon loading is completely in Chinese, and it has both Vue and React components. It feels kind of cluttered upon loading, having many links to other sections such as PrimeVue or ElementPlus. There are 74 total components in total with General (4), Layout (4), Data Display (19), Data Entry (21), Feedback (10), Navigation (6), Other (10).

Arco Design is the design system created by ByteDance, the parent company of TikTok/Douyin. It was open-sourced to provide a comprehensive design solution that bridges the gap between design and development. It competes directly with Ant Design in the enterprise space, reflecting the internal tools used to build ByteDance's massive portfolio of products. The Vue version was released in 2021. The repository has approximately 1,844 commits.
https://github.com/arco-design/arco-design-vue
https://arco.design/vue/en-US/docs/start

2.2.12 TDesign

The loading page is a bit slow likely as it is hosted in Chinese servers. A mix of both English and Chinese appears on the main page, with some logos and attestations. There seem to be web and mobile versions, along with Vue 2/Vue 3, WeChat Mini Programs, and a React version too. The components are Base (3), Layout (4), Navigation (10), Input (21), Data Display (24), and Notifications (8) with total 70.

TDesign is the enterprise design system from Tencent. Given Tencent's dominance in the Chinese market (WeChat, QQ, Gaming), TDesign was created to unify the user experience across their vast array of internal and external applications. It emphasizes consistency across different platforms, specifically catering to the WeChat Mini Program ecosystem alongside standard web frameworks. The Vue Next (Vue 3) version was released in 2021. The repository has over 3,500 commits.

https://github.com/Tencent/tdesign-vue-next
https://tdesign.tencent.com/vue

2.2.13 Flowbite Vue 3

A minimalistic yet well-styled VitePress site appears. This includes only the components and they are categorized Components (25), Form (9), Typography (5), and Utils (2) for a total of 41.

Flowbite Vue is an open-source library created by Themesberg. It is essentially a Vue-specific wrapper around Flowbite, which is a popular component library for Tailwind CSS. It was released in 2021. The repository has over 500 commits.

https://github.com/themesberg/flowbite-vue
https://flowbite-vue.com/

2.3 Analysis of Libraries

2.3.1 Initial aesthetic impressions

As a general rule, clear design is associated with clear abstractions. Using software with unclear abstractions present additional cognitive load: there may be multiple ways to do things, the instructions for how to do things may be unclear, or the syntax is messy and inefficient.

There are some exceptions: many low-level programming languages and libraries often use mdBook, which creates not exactly what one would call a

Table 2.1: Chronological Overview of Vue.js UI Libraries

Library	**Commits**	**Year**	**Author / Company**	**Country**	**Total**
Quasar	15,078	2015	Razvan Stoenescu	Romania	70
Vuetify	17,564	2016	John Leider	USA	78
Ant Design	5,302	2017	Tangjinzhou (Ant Group)	China	68
PrimeVue	14,184	2019	PrimeTek Informatics	Turkey	91
Naive UI	9,178	2019	TuSimple (07akioni)	China	79
ElementPlus	7,084	2020	Community (Eleme)	China	81
Arco Design	1,844	2021	ByteDance	China	74
TDesign	3,500	2021	Tencent	China	70
Flowbite	500	2021	Themesberg	Romania	41
shadcn-vue	809	2023	Mujahid Anuar	Malaysia	63
Nuxt UI	4,937	2023	NuxtLabs	France	119
Radix Vue	1,500	2023	Mujahid Anuar	Malaysia	43
Reka UI	1,775	2024	UnoVue	Malaysia	44

well-proportioned or modern design, but it is clarid and distinct in its own way in the way antiques have charm while modern grey-laminate flooring housing does not. Despite its age, it represents intelligent design.

For these reasons, the above design system libraries are first evaluated on their presentation. Furthermore, as developers will constantly scan back and forth between documentation and app, having an ugly website with annoyances such as unnecessary animations or page loads will make discerning developers displeased.

Because Quasar is not just a component library or design system but meant to develop multiple types of apps, I am eliminating it. Vuetify features an annoying fade animation when clicking between pages. I don't recommend TDesign because the page is slow to load and has internationalization problems, for example, the "Guideline" section Tab was fully in Chinese despite selecting the English locale. The other libraries: ElementPlus, shadcn-vue, Nuxt UI, Reka UI, Ant Design Vue, and Arco Design seem to fall somewhere in the middle. Reasonably good, but they extend their domain to many other things such as Vue 2 and React, trading clarity and conciseness for expansive coverage. Flowvite Vue 3, Radix UI, and Naive UI though simple, run the opposite risk of not having the specific component needed in complex projects but represent a good starting point due to simplicity.

2.3.2 Analysis of Categorizations

Based on the table, a clear consensus exists around a few core concepts, even if the exact naming differs.

Category Label	PrimeVue	ElementPlus	Naive UI	Ant Design	Arco Design	TDesign	Flowbite	Vuetify	Nuxt UI	Reka UI
Data Entry / Form	•	•	•	•	•	•	•	•	•	•
Data Display / Data	•	•	•	•	•	•		•	•	
Navigation / Menu	•	•	•	•	•	•		•	•	
Feedback / Messages	•	•		•	•	•		•		
General / Basic		•	•	•	•	•				•
Layout			•	•	•	•			•	
Other / Misc	•	•	•	•	•					
Button	•									
Chart	•									
Containment								•		
Element									•	
Components							•			
Page									•	
Selection								•		
Overlay	•								•	
Panel	•									
File	•									
Pickers / Dates								•		•
Images / Media	•							•		
Typography							•			
Config / Providers		•	•			•		•	•	

Table 2.2: Taxonomy Comparison: Component Categorization Labels

- **Data Entry / Form:** This is the most universal category, present in every single listed library that uses categorization. It represents all components used for user input (inputs, selects, checkboxes, etc.).
- **Data Display:** This is the second most common category, appearing in almost every library. It covers components meant to present information to the user (tables, lists, tags, avatars, etc.).

- **Navigation / Menu:** Also extremely common, this category groups components used for moving around an application (menus, breadcrumbs, tabs, pagination).
- **General / Basic:** Most libraries have a foundational category for fundamental building blocks like buttons (though PrimeVue separates them), icons, and links.
- **Layout:** A common category for components that structure the page, such as grids, dividers, and spacers.

2.3.3 Unique or Less Common Categories

These categories highlight the different philosophies or specializations of each library:

- `PrimeVue` has the most granular categorization, with many groups that other libraries merge into broader categories. `Panel`, `Overlay`, `File`, `Chart`, `Media`, and `Button` are all separated into their own top-level categories, which is unique among this set. `Reka UI` takes the opposite approach, grouping things into only 3 top-level categories. Some such as `shadcn-vue` only sort components alphabetically, having no categories at all.
- `Flowbite Vue 3` is the only library with a dedicated **Typography** category.
- **Feedback / Messages** (or *Notifications* in `TDesign`) is a fairly common concept for alerts, modals, and toasts, though `Naive UI` and `Flowbite` don't specify it at the top level.
- **Configuration / Config** appears in `ElementPlus`, `Naive UI`, and `TDesign` for global configuration providers.
- **Other / Misc / Utils** are common catch-all categories for components that don't fit neatly elsewhere.

2.3.4 Nomenclature Patterns and Cultural Distinctions

A closer examination of category labels reveals a distinct linguistic split that aligns partially with the geopolitical origins of the libraries.

There is a specific "Enterprise" nomenclature predominant in major Chinese commercial libraries (Ant Design, Arco, TDesign, Naive UI), which favors precise, noun-heavy descriptors like "Data Entry" and "Data Display." In contrast, Western-origin or community-led libraries (PrimeVue, Nuxt UI, Vuetify) tend to use shorter, functional terms like "Form" and "Data."

Data Entry vs. Form

This category represents the most distinct divergence in naming conventions.

Data Entry / Data Input
: Used almost exclusively by Chinese enterprise-backed libraries. This likely reflects a B2B "management system" philosophy where the user is an operator entering data.

- **Ant Design Vue** (Data Entry)
- **Arco Design** (Data Entry)
- **Naive UI** (Data Input)
- **TDesign** (Input)

Form / Form Inputs
: Used by Western libraries and the community-driven ElementPlus. This terminology aligns closer to the HTML `<form>` tag and standard web development vernacular.

- **PrimeVue** (Form)
- **Nuxt UI** (Form)
- **Reka UI** (Form)
- **Flowbite** (Form)
- **Vuetify** (Form Inputs and Controls)
- **ElementPlus** (Form) — *Notably acts as a bridge, being Chinese-origin but using Western naming conventions.*

Data Display vs. Data

Data Display
: Again, the explicit naming convention is found in the Chinese grouping.

- **Ant Design Vue**

- **Arco Design**
- **Naive UI**
- **TDesign**

Data
A shorthand found in libraries that prioritize conciseness.

- **PrimeVue**
- **ElementPlus**
- **Nuxt UI**

Data and Display
Vuetify combines these concepts into a single verbose category.

Feedback vs. Messages vs. Notifications

This category handles modals, toasts, and alerts, showing the most variance in naming.

Feedback
The most common term across both regions for interactive responses.

- **ElementPlus**
- **Vuetify**
- **Ant Design Vue**
- **Arco Design**

Messages
Specific to **PrimeVue**, likely deriving from the Java/JSF heritage of PrimeFaces where "FacesMessages" were a core concept.

Notifications
Specific to **TDesign**.

Navigation vs. Menu

Navigation
The standard term used by the vast majority of libraries (ElementPlus, Nuxt UI, Vuetify, Naive UI, Ant Design, Arco, TDesign).

Menu

Specific to **PrimeVue**. This is an interesting deviation, as PrimeVue separates standard navigation elements (like Breadcrumbs) into a "Menu" category, implying a specific UI widget focus rather than an abstract navigation concept.

2.3.5 On Design System Infrastructure

I will use the term "infrastructure" as the catch-all term for important factors such as Typescript support, tree-shaking, installation method, and accessibility.

Installation methods are currently: install a npm package and import, use a CDN (no tree-shaking, which unused components are bundled and reduces performance), or copy-paste, popularized by shadcn, (which can be done manually or with the CLI).

Typescript is necessary because when configuring the component, Intellisense tells you what options you can put and if you put any invalid options into it.

Accessibility refers to focus management and focus visibility (blue ring around items) which involve keyboard navigation, ARIA states (which may specify when something is open or closed to a screen reader), minimum touch target size and color contrast, and reduced motion.

Finally is how styling is overriden in the component itself.

2.4 On categorization

The categorization of these components can take an extreme (listing a new category for each group of new similar components, as per PrimeVue), or it can take the opposite extreme of not categorizing things at all. It is certain that as a component library grows, small groups of 2-5 components will appear which bear no resemblance to the major categories, such as PrimeVue's Media category of Carousel, Galleria, Image, and ImageCompare. To sort things alphabetically would break up these small categories and make things unmanageable, so despite small categories making the top level more verbose I believe they are preferable once the component library passes a certain size.

2.4.1 A new component library taxonomic suggestion

When designing a taxonomy one must ensure it captures the essence of what is important and relevant. A climate taxonomy on average plant leaf size is likely to be a poor predictor of environmental landscapes. These features and attributes may be a numeric sliding scale, or they may be categorical from a finite set of options such as the Köppen-Geiger where one picks from climate, precipitation, and temperature severity during summer or winter (optional) to create a key such as ET (Polar + Tundra) or Cw (Temperate + Dry winter).

Human abstractions, like may things, may be precise or imprecise. The periodic table is an abstraction which accurately captures the nature of different elements: it is complete, the protons follow and incrementing order and they are grouped column-wise by similar properties. An incomplete abstraction would be something like: we have these metals, here we have gases, next we have the elements: fire, earth, water, and air.

Some may see the Köppen-Geiger scale and feel the lowercase letters (it uses both uppercase and lowercase letters to denote various climates) add to an imprecision and confusion because we are now indexing on another axis: letter capitalization. To remove what is necessary is essential and characteristic of clear thought and abstraction, for without it we confuse others with nonsensical abstractions we have created. For example, React's useState is confusing: it is used both for local component state and global state with useContext, while Vue separates refs (local state) and uses Pinia for global state. useEffect is used for mounting and unmounting a component, yet it involves passing in parameters to the function (what even is an effect?) rather than Vue's simple mount/unmount functions.

2-category classifications are quite common. One consistent theme is the idea of input, output, and internal state. We can draw a bounding box around humans and our brain (internal state), enumerate our senses, and list the potential actions. Terminals have stdin and stdout, as do computers have inputs and outputs. Shannon's Model of Communication also uses this input/output/internal state pattern.

The Linux Filesystem Hierarchy distinguishes between variable files (stored in /var/ and change frequently during normal system use such as logs, state information, caches) and static files (stored in a variety of directories) which are often binaries, configurations, and dependencies.

With inspiration from the above examples, in the taxonomy of web components we first have a display of information starting with the terminal, and this originally was only text. But frequently we use pictures and symbols to denote things, hence the first distinctive factor of component libraries should

be this TEXT/SYMBOLIC key.

Images, progress bars, and avatars would count as symbolic. Of course, many components are not either/or, but may combine both of those features, but these mixed components thus represent a higher-level feature than pure text or symbols.

Another taxonomic key that immediately stands out is the input/output distinction. Of inputs, either a click input, text input, or hover is possible, but since all inputs require some visual on the screen so as to allow someone to navigate an input, considering such existence as an "output" belies the true categorization of the components. Outputs should then be things that only display things on the screen and allow for no input at all: no clicks nor text entry nor hover.

Since components are likely to have multiple of these keys, the proper usage in a component library is not so much to group them into arbitrary categories from a multiplication of those keys but to perhaps have some sort of tab-search-multi group at the top so that one can find the proper components given their knowledge of the component's functionality. Artificial taxonomies just then become another thing people have to learn, getting them further away from what they need to build things.

The Data Entry/Data Display categories have similarities with the aforementioned proposed input/output grouping while Navigation, Menu, General, or Layout are examples of common categorical groupings. Labels such as Button, Chart, Components, Page, Selection, Overlay, Panel, Images, and so on are also categorical groupings, but they are less common.

A future design system categorization should thus reuse existing common taxonomic classifications but also consider a text/symbolic and input/output key.

2.5 On styling and overconstraints

A component library may come styled or headless with no styles. When it comes styled, it will often include features that represent additional common functionalities of that component: a button may include primary, secondary, icon positioning, loading states, or button size.

The installation method of design systems also matters greatly for styling. Shadcn was designed as a copy-paste solution while other methods are difficult to override, involving deep CSS selectors such as :deep() or !important and creating brittle, verbose code. Using tools like patch-package to override styles create version-dependent multiple-megabyte .patch files which must be

committed to the git repository.

Choosing which component library thus depends upon the needs of the user. If it is a cover page, marketing page, or for a use where aesthetics absolutely matter such as a museum or product website, yet amount of total components is low, then headless components that are easily styled are the best choice. If it is a complex dashboard that requires something functional over aesthetic, choosing something that has been established and has numerous types of components is a higher priority, so Ant Design Vue may be prefered over Reka UI (68 vs 44 components). The former also has previews of different component states, a lack of which can be annoying because you'd have to add any prop to your app to see what it really does.

Thus, another table is given for the aforementioned component libraries on how many components they have, their installation method, and what styling method is used for them.

Table 2.3: Infrastructure, Installation, and Styling Overrides of Vue Libraries

Library	Total	Installation	Override / Styling Method	Component URL
PrimeVue	91	npm package	Pass-Through (PT) props or Unstyled	https://primevue.org/uikit
ElementPlus	81	npm package	SCSS Vars / Deep Selectors	https://element-plus.org/en-US/component/button.html
Naive UI	79	npm package	JS Config Provider / CSS Vars	https://www.naiveui.com/en-US/os-theme/components/button
Ant Design Vue	68	npm package	Less Variables	https://antdv.com/components/overview
Arco Design	74	npm package	Less Variables	https://arco.design/vue/component/button
TDesign	70	npm package	CSS Variables	https://tdesign.tencent.com/vue/components/overview
Nuxt UI	119	Nuxt Module	AppConfig / Tailwind Classes	https://ui.nuxt.com/components
shadcn-vue	63	Copy-Paste (CLI)	Direct Code Modification (Tailwind)	https://www.shadcn-vue.com/docs/components/accordion
Flowbite Vue	41	npm package	Tailwind Classes	https://flowbite-vue.com/components/buttons/button/
Radix Vue	43	npm package	Headless	https://www.radix-vue.com/components/accordion.html
Reka UI	44	npm package	Headless	https://reka-ui.com/docs/components/accordion

Chapter 3

Generation of components using LLMs

In this section, a variety of primitive components will be generated against a spec and evaluated. First, the humble button.

3.1 Analysis of Button

In this section, the Button documentation is analyzed from the perspective of numerous design systems that we narrowed down in the last section. Contrary to initial thought of which libraries may be good to use, the minimalist appearing websites seem to be lacking significant functionality for Button, which may generalize to other components.

- **PrimeVue:** An import statement is shown at the top. Examples are good and clear. The label is passed in with a prop, icons with left/right and top/bottom positioning is possible. Icons seem to be PrimeVue specific and passed in as a class. Loading, link, disabled states exist. Primary/secondary categorization distinction. Icon only, badge group, and a variety of styling options exist.

- **ElementPlus:** Numerous primary/secondary/icon only typing exists. Disabled, state, and link states exist. Code examples are not immediately obvious as you have to click "view source", edit on Github is an error page, and copy code has to be pasted, so it is not immediately clear what some of the other API attributes do. Due to the addition of an API section at the bottom and PrimeVue lacking it, ElementPlus actually has more features despite them not all being shown.

- **Naive UI:** Code buttons next to the examples similar to ElementPlus. Initial impression is that it is similar to the other examples.
- **Ant Design Vue:** Style seems similar to the above both in the code example buttons and button functionality.
- **TDesign:** Documentation style is different from all of the others. There are three tab sections rather than one long and scrollable page. Functionality seems mostly similar.
- **Nuxt UI:** Style is more similar to PrimeVue, there is a button at the top of the documentation page which allows copying markdown or pasting into an LLM. This has an API section unlike PrimeVue, and a Changelog.
- **shadcn-vue:** Copy page as Markdown button appears, and the API reference is very small comparatively. The page is not very dense, making it unpleasant to read.
- **Flowbite Vue 3:** The amount of features is markedly reduced compared to other libraries. The button props seem to primarily be for styling. Any mention of icon positioning are not there, and it is called "slot" instead.
- **Radix Vue:** No button component exists.
- **Reka UI:** No button component exists.

3.2 Generating the Button

To test the capabilities of LLMs, rigor is needed. First, the button component was thoroughly evaluated. Next comes the spec, then comes the generation, and finally the evaluation. It is roughly a four step process, where the last two to three can be done by LLMs.

- Analyze a common component to determine what it should have
- Write out a spec for it
- Generate it
- Evaluate it against the spec

The first step can be considered loading the human brain with information, and every further step is to evaluate the created materials for accuracy. Thus, progress happens to the limit of human perception. The prompt for the spec is as follows:

> `Use the Vue 3 Composition API, WAI-ARIA accessibility guidelines, and TypeScript to generate a headless element in a component library. The element is Button: it should have options for an icon with variable positioning to all sides of the element and a loading state.`

Although other system prompts are generally multi-line bullet points in Markdown, this basic human-written one serves as a rough, simple guideline starting from the beginning.

3.3 Evaluation of the Button

After generation of the button, two different methods of displaying it were displayed. First, in Storybook and the second just in Vue directly.

What separates high quality human abstractions from machine-generated sloppiness is the degree of "sharpness" and precision inherent in high quality human work.

I then fed this thesis and component.library's website into the button to tell it to generate a new spec for the thesis.

3.4 AI-generated Spec

```
### LLM Instructions to Generate a Button Component

**Role:** You are an expert frontend developer specializing in creating accessible, reusable, and robust UI components. Your work prioritizes semantic markup, WAI-ARIA compliance, and a clean, developer-friendly API.

**Task:** Generate a headless **Button** component for a Vue 3 application. The component should be a single `.vue` file using the Composition API (`<script setup>`) and TypeScript. It must be unstyled by default but architected for easy customization via props and slots, intended to be styled by a utility-first CSS framework like Tailwind CSS or with scoped CSS.
```

```
6
7  Adhere strictly to the following requirements, which are based
       on best practices for modern web components.
8
9  ---
10
11 ### **Component Specification: `Button.vue`**
12
13 #### **1. Core Markup & Semantics**
14 - The root element **must** be a native `<button>` element. Do
       not use `<a>`, `<div>`, or `<span>`.
15 - It must accept a `type` attribute with possible values `'
       button'`, `'submit'`, or `'reset'`. The default `type`
       should be `'button'` to prevent accidental form submissions
       , which is a safer default than the browser's native `'
       submit'`.
16
17 #### **2. Props API (Define with `defineProps`)**
18 The component's interface should be strongly typed with
       TypeScript.
19
20 - `type`: `(type: 'button' | 'submit' | 'reset') => { default:
       'button' }`
21 - Defines the button's behavior.
22 - `disabled`: `(type: Boolean) => { default: false }`
23 - Disables the button from being interactive. Should apply the
       `disabled` attribute to the underlying `<button>`.
24 - `loading`: `(type: Boolean) => { default: false }`
25 - Puts the button in a loading state. When `true`, the button
       must also be disabled to prevent further clicks.
26 - `iconPosition`: `(type: 'left' | 'right') => { default: 'left
       ' }`
27 - Determines where the icon is rendered relative to the button'
       s text label.
28
29 #### **3. Slots (Define with `<slot>`)**
30 Use slots to make the component's content flexible and
       composable.
31
32 - **`default` slot:** For the primary text content of the
       button (e.g., "Submit", "Delete").
33 - **`icon` slot:** A named slot for passing an SVG or an icon
       component. This slot should only be rendered if content is
       provided for it.
34 - **`loader` slot:** A named slot for a loading indicator (e.g
       ., a spinner). This slot should only be rendered when the `
       loading` prop is `true`, and it should replace both the `
       icon` and `default` slots.
```

```
35
36 #### **4. Behavior and Accessibility (WAI-ARIA)**
37 - When `loading` is `true`:
38 - The button must have the `aria-busy="true"` attribute.
39 - The button must have the `disabled` attribute.
40 - The `loader` slot content should be displayed, and all other
    content (text, icon) should be hidden.
41 - When `disabled` is `true`:
42 - The button must have the `aria-disabled="true"` attribute in
    addition to the native `disabled` attribute.
43 - The component must be fully keyboard-accessible by default,
    leveraging the native `<button>` element's focusability and
     activation (via `Space` and `Enter` keys).
44 - **Note for consumer:** While not implemented in the component
     itself, remind the user that if the button contains *only
    an icon* and no visible text, they are responsible for
    providing an `aria-label` for screen reader users.
45
46 #### **5. Styling Philosophy**
47 - The component should be **headless**. Do not include any CSS
    in the `<style>` block except for what is absolutely
    necessary for functionality (e.g., a class to visually hide
     text for screen readers if needed).
48 - The template should use a logical structure that allows for
    easy styling with utility classes passed via the `class`
    attribute from the parent. For example, use `<span>`
    wrappers for the icon and text to allow individual styling.
49
50 #### **6. Final Output**
51 - Generate the complete code for the `Button.vue` Single File
    Component.
52 - Include the `<template>`, `<script setup lang="ts">`, and an
    empty `<style scoped>` block.
53 - Provide a brief example of how to use the component in a
    parent, showing a standard button, a button with an icon,
    and a button in its loading state.
```

Listing 3.1: LLM Prompt for Generating a Button Component

3.5 Test

Despite generating a much longer spec, it seems the AI was not able to

Chapter 4

Summary and Conclusion

Through the introduction significant understanding of the world, the development of technologies, and its political nature were revealed. Next, a deep dive into numerous component libraries and the rationale for choosing Vue over React was made. The cultural influences and categorical grouping of components were discussed, and a new taxonomic key created. Generation of common components using LLMs were tested to measure the approach of bottom-up approach of creating technologically repeatable subunits quickly compared to paring down a large existing component library.