The State of Frontend Design Engineering 2026: Technical Architectures of the Modern Web

1. Introduction: The Convergence of Design and Engineering

The distinct disciplines of interface design and frontend engineering, once separated by the "handoff" chasm, have in 2026 coalesced into a singular, high-leverage role: the Design Engineer. This evolution was not merely a change in title but a fundamental shift in the technical architecture of the web. The static, component-library-driven development of the early 2020s has given way to a dynamic, physics-based, and mathematically fluid web. This report analyzes the technical foundations of this new era, dissecting the engineering patterns championed by industry leaders like Vercel, Linear, and Stripe, and validated by the avant-garde of Awwwards "Site of the Day" winners.

The mandate for the modern interface is clear: it must feel "native." This descriptor, once vague, now corresponds to specific engineering metrics—60fps (or 120fps on ProMotion displays) consistency, non-linear spring physics for interactions, sub-pixel accurate typography that respects user accessibility settings, and layout systems that adapt fluidly rather than snapping rigidly between breakpoints. Achieving this requires a deep understanding of the browser's rendering engine, the mathematics of interpolation, and the emerging capabilities of the CSS Object Model (CSSOM).

In this report, we rigorously examine the implementation details of the "premium" web stack. We explore how CSS Subgrid has enabled the complex, asymmetrical balance of Bento grids; how variable font technology and metric overrides have solved the "Flash of Unstyled Text" (FOUT) without compromising layout stability; and how the next generation of CSS primitives—shape(), sibling-index(), and scroll-driven animations—are allowing engineers to offload complex JavaScript logic to the browser's compositor thread. This is a technical blueprint for the state of the art in 2026.

---

2. Fluid Typography Engineering: The Mathematics of the 2026 Typesystem

Typography on the modern web has moved beyond simple breakpoint-based media queries. The standard for 2026 is "Fluid Typography," a system where type scales continuously and

mathematically between viewport extremes. However, the implementation of this system has matured from simple calc() equations to robust, accessible architectures that prevent layout shifts and respect user overrides.

2.1 The Evolution and Pitfalls of CSS clamp()

The cornerstone of fluid typography is the CSS clamp() function, which interpolates a value between a minimum and maximum based on a preferred change rate. A typical 2024 implementation might have looked like font-size: clamp(1rem, 5vw, 2rem). By 2026, engineering analysis has exposed significant flaws in this naive approach, particularly regarding accessibility and browser zoom behavior.[1]

2.1.1 The Accessibility Failure of Pure Viewport Units

A critical engineering constraint in 2026 is the handling of browser zoom. When a user zooms in via the browser (Cmd/Ctrl +), the viewport dimensions (viewport width, vw) often remain unchanged while the pixel density increases, or the browser emulates a smaller viewport. Pure vw based typography fails WCAG 1.4.4 (Resize text) because the text does not scale up proportionally with the rest of the page content.[1]

The 2026 Solution: Relative Unit Mixing

To ensure accessibility, modern fluid equations must mix relative units (rem) with viewport units. The rem unit ensures that the user's base font size preference (usually 16px, but variable) is respected as a multiplier.

The engineered formula for a robust clamp function is:

$$f(x) = \text{clamp}(\text{min\_size}, \text{slope} \times \text{viewport} + \text{base\_intersect}, \text{max\_size})$$

In CSS, this manifests as a calculation that anchors the scaling slope to a rem value. For example, instead of 5vw, the preferred value becomes calc(1rem + 2vw). This ensures that even if vw remains static during a specific zoom behavior, the 1rem component expands, forcing the text to scale.[3]

2.1.2 Container Query Units (cqw) over Viewport Units (vw)

With the ubiquitous support of Container Queries in 2026, the industry has shifted away from vw for component-level typography. Viewport units equate the text size to the browser window, which breaks modularity when a component is placed in a sidebar versus a main content area.

The modern standard utilizes cqw (Container Query Width). This allows typography to be fluid *relative to its parent container*, enabling true "write once, deploy anywhere" component

architecture.

Implementation Pattern:

CSS

```css
.card {
  container-type: inline-size;
}

.card__title {
  /* Scales based on the card's width, not the screen's width */
  font-size: clamp(1.25rem, 5cqw + 1rem, 2.5rem);
}
```

This approach decouples the typography from the global context, aligning with the component-driven architecture prevalent in frameworks like Next.js and React.[3]

2.2 Font Loading Strategies and FOUT Prevention

The "premium" feel of sites like Stripe and Vercel relies heavily on the elimination of "Flash of Unstyled Text" (FOUT) and "Cumulative Layout Shift" (CLS). In 2026, the engineering focus has shifted to advanced font metric overrides and loading strategies that prioritize visual stability.

2.2.1 The Geist Loading Strategy

Vercel's introduction of the Geist font highlighted a sophisticated loading strategy designed to eliminate layout shifts completely. The core of this strategy involves matching the fallback font's metrics to the web font's metrics so precisely that the swap is imperceptible.[5]

When a custom font loads, it often has different aspect ratios, x-heights, and baselines than the system fallback (e.g., Arial or San Francisco). This causes the text to reflow, pushing content down and degrading the user experience.

2.2.2 Metric Overrides: size-adjust, ascent, and descent

To combat this, 2026 engineering pipelines utilize CSS descriptors within the @font-face rule to force the fallback font to occupy the exact same physical space as the loading web font.

Technical Implementation:

1. **size-adjust**: Scales the glyphs of the fallback font to match the width of the web font.

2. **ascent-override**: Adjusts the height above the baseline.
3. **descent-override**: Adjusts the depth below the baseline.
4. **line-gap-override**: Normalizes line spacing.

Code Artifact:

CSS

```css
/* Fallback font adjusted to match 'Geist Sans' metrics */
@font-face {
  font-family: 'Geist-Fallback';
  src: local('Arial');
  ascent-override: 96%;  /* Adjusted to match Geist */
  descent-override: 20%; /* Adjusted to match Geist */
  size-adjust: 104%;     /* Scales Arial width to match Geist */
}

body {
  font-family: 'Geist Sans', 'Geist-Fallback', sans-serif;
}
```

Tools and frameworks (like next/font) now automate the calculation of these values at build time, ensuring that the fallback font is mathematically identical in bounding box size to the custom font.[7]

2.2.3 Preloading and font-display

While font-display: swap was the recommendation of the early 2020s, the 2026 standard for high-performance marketing sites (like Linear's) often leans toward font-display: optional for the initial load, or highly optimized swap with metric overrides.

font-display: optional gives the browser a short window (100ms) to load the font. If it fails, the browser uses the fallback font for the *rest of the page view*, preventing any layout shift or text flash during reading. The web font is then downloaded in the background and cached for the next navigation, ensuring subsequent page loads are instant and styled correctly.[5]

---

3. Advanced Layout Systems: The Bento Grid and CSS Subgrid

The "Bento Grid," popularized by Apple's promotional videos and adopted by Linear and specialized SaaS dashboards, has become the dominant layout pattern for dense information display in 2026. This layout is characterized by a strict grid of bento-box-like compartments, rounded corners (typically 20-24px), and a unified spacing rhythm.[11]

Implementing this layout responsively, without breaking the visual alignment of internal elements across different cards, requires the use of **CSS Subgrid**, a feature that has reached maturity and widespread support by 2026.

3.1 The Architecture of a Subgrid System

The primary challenge in Bento grids is aligning content *inside* separate cards with each other. For example, a header in "Card A" should visually align with a header in "Card B," even if they are in different DOM containers. Traditional CSS Grid treats each card as an independent layout context, making this cross-card alignment impossible without "magic numbers" (fixed heights).

CSS Subgrid (grid-template-columns: subgrid;) solves this by allowing nested children to participate in the parent's grid definition.

3.1.1 Implementation Details

In a 2026 production environment, the Bento grid is typically defined as a complex 12-column or 24-column parent grid. The child cards span multiple columns and rows, but critically, they inherit the grid tracks for their own internal layout.[14]

Code Artifact:

CSS

```
.bento-container {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  grid-template-rows: auto;
  gap: 24px;
}

.bento-card {
  grid-column: span 4; /* Spans 4 columns of the parent */
  display: grid;
  grid-template-columns: subgrid; /* Inherits the 4 columns it spans */
```

```
  grid-template-rows: subgrid;  /* Optional: aligns internal rows */
}
```

This architecture ensures that if the parent grid changes (e.g., a column becomes narrower on a smaller screen), every element inside every card adjusts synchronously. It allows for "Stripe-quality" layouts where a feature list in one card aligns pixel-perfectly with an image in an adjacent card.[12]

3.2 Responsive Bento Patterns

The rigidity of a Bento grid poses challenges for responsive design. The 2026 standard for adapting these grids to mobile involves two primary strategies:

1. **Reflow to Feed:** On mobile devices, the grid collapses into a single column (grid-template-columns: 1fr). However, to maintain the "Bento" feel, the aspect ratios of the cards are often preserved or adjusted using the aspect-ratio property, rather than letting content dictate height entirely.
2. **Dense Packing:** The grid-auto-flow: dense property is utilized to fill gaps. If a large 2x2 card wraps to a new line, smaller 1x1 cards can backfill the empty space, maintaining the "solid wall" aesthetic of the Bento box.[16]

The Apple Methodology: Apple's implementation often relies on a fixed set of aspect ratios. Boxes are rarely random sizes; they are usually 1x1 (square), 2x1 (landscape), or 1x2 (portrait). This modularity allows the grid to be rearranged mathematically rather than arbitrarily. Javascript libraries like Masonry are largely deprecated in 2026 favor of native CSS Grid capabilities, which offer superior performance and less layout thrashing.[13]

3.3 Visual Hierarchy and "Noise" Reduction

A key insight from the analysis of Linear and Apple designs is the management of visual density. A Bento grid works because it compartmentalizes complexity. Each box contains a single idea. Engineering this involves strict content limiting—using CSS line-clamp to truncate text and object-fit: cover to ensure media always fills its assigned cell without distorting the strict grid lines.[12]

---

4. Physics-Based Micro-interactions: The "Native" Web Feel

The distinguishing feature of high-end frontend engineering in 2026 is the abandonment of time-based animation (Duration/Easing) in favor of physics-based animation (Springs). This shift is driven by a desire to emulate the tactile responsiveness of native iOS/Android interfaces.

4.1 The End of Duration

Traditional CSS transitions rely on a fixed duration (e.g., 0.3s ease-out). This creates a disconnect between the user's input velocity and the interface's response. If a user flicks an element quickly, a fixed-duration animation feels sluggish. If they move it slowly, the animation feels rushed.

Spring Physics eliminates duration as a primary variable. Instead, motion is defined by physical properties:

- **Stiffness (Tension):** The rigidity of the spring. Higher tension = faster, snappier movement.
- **Damping (Friction):** The force opposing the motion. Lower damping = more oscillation (bounciness).
- **Mass:** The weight of the object. Heavier objects accelerate and decelerate slower.[17]

4.2 The Golden Configuration for Web Springs

While "perfect" values are subjective, analysis of Vercel and iOS system behaviors points to a "Golden Configuration" often used as a baseline for premium web interactions.

The "Snappy but Smooth" Config:

- **Stiffness:** 150 - 200
- **Damping:** 15 - 20
- **Mass:** 0.1 - 0.5 (Low mass prevents a "sluggish" start)

This configuration creates a motion that feels immediate (responsive) but settles softly without excessive oscillation, mimicking the "critical damping" point used in Apple's SwiftUI.[17]

4.3 Implementing the Magnetic Button

A quintessential micro-interaction of the 2026 Design Engineer stack is the "Magnetic Button"—a button that physically attracts the mouse cursor when it hovers nearby. This effect breaks the boundary of the standard DOM box model, suggesting the UI element is a physical object influenced by the user's presence.

4.3.1 Mathematical Implementation

The implementation relies on calculating the vector distance between the cursor and the center of the button element.

$$\Delta x = \text{Cursor}_x - (\text{ButtonLeft} + \text{Width}/2)$$

$$\Delta y = \text{Cursor}_y - (\text{ButtonTop} + \text{Height}/2)$$

This delta is not applied 1:1. Instead, it is dampened (e.g., multiplied by 0.3) so the button moves *towards* the cursor but doesn't chase it completely.

4.3.2 The Performance Stack: Framer Motion vs. React Spring

In 2026, **Motion (formerly Framer Motion)** has established itself as the standard library for these interactions due to its superior developer experience (DX) and hardware-accelerated layout projection.[22]

Code Strategy (Motion):

JavaScript

```
// Using a ref to track the element and state for position
const ref = useRef(null);
const [position, setPosition] = useState({x:0, y:0});

const handleMouse = (e) => {
  const { clientX, clientY } = e;
  const { height, width, left, top } = ref.current.getBoundingClientRect();

  // Calculate distance from center
  const middleX = clientX - (left + width/2);
  const middleY = clientY - (top + height/2);

  // Update position with dampening factor
  setPosition({ x: middleX * 0.2, y: middleY * 0.2 });
}

// Render with spring transition
<motion.div
  animate={{ x: position.x, y: position.y }}
  transition={{ type: "spring", stiffness: 150, damping: 15, mass: 0.1 }}
  onMouseMove={handleMouse}
  onMouseLeave={() => setPosition({x:0, y:0})}
>
  {children}
</motion.div>
```

This implementation uses the spring type to ensure that when the mouse leaves the button, it

snaps back to origin (0,0) with a physical bounce rather than a linear slide.[21]

Benchmark Insight: While libraries like **Motion One** (built on the Web Animations API) offer smaller bundle sizes (approx. 3.8kB vs Motion's ~85kB), the "Design Engineer" consensus favors Motion (Framer Motion) for complex, state-driven interactions like magnetic buttons because of its robust layout projection engine and declarative React API. For simple scroll animations, lighter libraries are preferred, but for tactile component physics, Motion remains the industry heavy lifter.[24]

---

5. Next-Gen CSS Animations: The Stutter-Free Marquee

The infinite marquee (a continuously scrolling strip of logos or text) is a staple of SaaS marketing pages. Historically, this was implemented using JavaScript to clone elements or complex CSS keyframes that often jittered on mobile devices. 2026 introduces a CSS-native, performant approach using new browser primitives.

5.1 The CSS shape() and sibling-index() Revolution

Advanced implementations in 2026 leverage the experimental-but-stabilizing CSS features shape() and sibling-index(). These allow elements to follow a path and offset their animation start times based on their DOM order without JavaScript loop calculations.[27]

The Technique:

1. **offset-path (via shape()):** Defines the trajectory of the marquee items. Unlike transform: translateX, using offset-path can sometimes offer better sub-pixel rendering on high-DPI displays.
2. **sibling-index():** This CSS function returns the integer index of an element among its siblings.
3. **sibling-count():** Returns the total number of siblings.

By combining these, the animation delay can be calculated dynamically in CSS:

CSS

```
.marquee-item {
  /* Calculate delay based on position in the list */
  animation-delay: calc( -1s * sibling-index() * var(--duration) / sibling-count() );
```

}

This ensures a perfectly distributed stream of elements regardless of how many items are added to the DOM.[27]

5.2 Performance Optimization: The Compositor Thread

To ensure the marquee is "stutter-free" (60fps/120fps), 2026 engineering dictates that *only* the transform and opacity properties be animated. Properties like left, margin, or background-position trigger Layout or Paint operations, causing CPU spikes and visible jitter (jank).

Hardware Acceleration Force: Historically, will-change: transform or transform: translateZ(0) was used to promote layers. In 2026, browsers are smarter, but explicit layer promotion is still a safeguard for infinite loops. Critically, object-fit and image decoding must be handled before the animation starts. Using loading="eager" on marquee images prevents the "pop-in" effect during the first cycle of the animation.[30]

---

6. Visual Effects and Shaders: The Cursor Follower

High-end interaction design often utilizes a custom cursor that modifies the content beneath it. The "Difference" blend mode cursor is a specific trend identified in Awwwards "Site of the Day" winners.

6.1 CSS mix-blend-mode: difference

This effect relies on the CSS compositing engine. A is fixed to the cursor position (usually via a portal or direct body injection) and styled with mix-blend-mode: difference.

- White cursor on White background = Black (Difference 0)
- White cursor on Black text = White (Inversion)

Engineering Challenge:

The naive implementation involves updating the top/left properties of the cursor div on every mousemove event. This causes layout thrashing.

The optimized 2026 approach uses **CSS Variables** or **Direct Transform Updates**:

1. **CSS Variables:** Update --x and --y variables on the body or a wrapper.

.cursor { transform: translate3d(var(--x), var(--y), 0); } ``` 2. **RequestAnimationFrame (rAF):** JavaScript should not update the DOM directly on every mouse event. Instead, mouse coordinates should be stored in a variable, and a requestAnimationFrame loop should apply the visual update. This decouples the input sampling rate (which can vary) from the screen

refresh rate (typically 60Hz or 120Hz), ensuring smooth motion without "slipping".[21]

6.2 Linear Interpolation (Lerp) for Fluidity

A raw cursor follower feels robotic. To add "weight" (a key characteristic of the Linear/Vercel aesthetic), engineers apply Linear Interpolation (Lerp) to the cursor's movement.

$$P_{current} = P_{current} + (P_{target} - P_{current}) \times 0.1$$

The cursor follows the mouse but with a slight delay (0.1 factor), smoothing out jittery hand movements and creating a cinematic, fluid feel.[32]

---

7. Engineering Insights from Industry Leaders

7.1 The "Design Engineer" at Vercel and Linear

The role of the Design Engineer at companies like Vercel and Linear has redefined the technical requirements of frontend development. It is no longer sufficient to match a Figma mockup; the engineer must own the *execution* of the interaction.[33]

- **Vercel:** Heavily utilizes "delighters"—small, non-critical animations that enhance perceived quality (e.g., the glow on a card hover or the specific spring bounce of a modal). Their engineering culture emphasizes "Iterate to Greatness," often prototyping interactions directly in code to test the "feel" which cannot be simulated in Figma.[33]
- **Linear:** Known for the "Linear look"—dark mode, subtle gradients, and extremely performant, keyboard-driven interfaces. Their engineering relies on custom WebGL shaders for "glow" effects and highly optimized CSS for layout, avoiding heavy frameworks where native primitives suffice.
- **Stripe:** Pioneers of the "Connect" aesthetic, heavily utilizing 3D (Three.js/React Three Fiber) and complex gradient meshes. Their "marquee" implementations often use WebGL for distortion effects, going beyond simple CSS transforms.[34]

7.2 The Rise of Systematized Motion

Just as Design Systems standardized color and typography in the 2010s, 2026 sees the standardization of **Motion Tokens**. Companies are defining standard spring configurations (e.g., --ease-elastic-1, --spring-stiff) in their CSS variables, ensuring that every button click and page transition shares the same physical "brand feel".[35]

---

8. Emerging Trends in Visual Engineering

8.1 Shader-Based UI Elements

Moving beyond the DOM, "Site of the Day" winners increasingly employ WebGL shaders for UI elements that were previously static. This includes "noise" textures that animate (film grain) to reduce color banding in dark mode gradients, and "glassmorphism" effects that use real-time background blurring with chromatic aberration, simulated via fragment shaders rather than simple CSS backdrop-filter.[33]

8.2 Scroll-Driven Animations

The W3C Scroll-driven Animations specification has gained traction, allowing animations to be linked directly to the scroll offset of a container or the page. This enables parallax effects, reading progress bars, and "reveal" animations to be declarative in CSS, removing the need for IntersectionObserver in many cases and running entirely off the main thread.[36]

---

9. Conclusion: The Native Web Era

The frontend engineering landscape of 2026 is defined by a convergence of web technologies towards "native" performance and fidelity. The trends analyzed—fluid typography that respects physics and accessibility, Bento grids that leverage advanced CSS subgrids, and interaction models based on spring dynamics—all point to a singular goal: erasing the distinction between a "website" and a "native application."

For the Design Engineer, mastery of these domains is not optional. It requires a deep understanding of the browser's rendering engine (to prevent FOUT and jank), a grasp of physics (to tune springs), and a command of modern CSS layouts (Subgrid). The result is a web that is not only visually stunning but mathematically robust and accessibly engineered.

Summary of Key Recommendations for 2026 Implementation:

Domain	Legacy Approach (2020-2024)	2026 Standard
Typography	px or simple clamp()	clamp() with rem mixing + cqw units
Font Loading	font-display: swap	size-adjust overrides + optional or optimized swap
Grid Layout	Masonry.js / Nested Flexbox	CSS Grid + subgrid + grid-auto-flow: dense

Animation	Duration + Easing (Bezier)	Spring Physics (Mass/Stiffness/Damping)
Micro-interaction	CSS :hover	Magnetic Cursor (React/Motion) + Lerp smoothing
Marquee	JS Cloning / keyframe jank	shape() + sibling-index() + Compositor only

Referências citadas

1. Responsive Typography That Actually Works: Beyond font-size: clamp() | by Roberto Moreno Celta, acessado em fevereiro 16, 2026, https://robertcelt95.medium.com/responsive-typography-that-actually-works-beyond-font-size-clamp-acf592b79774
2. CSS Fluid Typography: A Guide to Using clamp() for Scalable Text - DEV Community, acessado em fevereiro 16, 2026, https://dev.to/devyoma/css-fluid-typography-a-guide-to-using-clamp-for-scalable-text-293o
3. Responsive and fluid typography with Baseline CSS features ..., acessado em fevereiro 16, 2026, https://web.dev/articles/baseline-in-action-fluid-type
4. Container Query Units and Fluid Typography - Modern CSS Solutions, acessado em fevereiro 16, 2026, https://moderncss.dev/container-query-units-and-fluid-typography/
5. Loading web fonts - Vercel Examples, acessado em fevereiro 16, 2026, https://solutions-loading-web-fonts.vercel.app/
6. How to use Vercel's Geist Font in Next.js? - Peerlist, acessado em fevereiro 16, 2026, https://peerlist.io/blog/engineering/how-to-use-vercel-geist-font-in-nextjs
7. Improved font fallbacks | Blog - Chrome for Developers, acessado em fevereiro 16, 2026, https://developer.chrome.com/blog/font-fallbacks
8. CSS size-adjust for @font-face | Articles - web.dev, acessado em fevereiro 16, 2026, https://web.dev/articles/css-size-adjust
9. Reducing layout shift with custom fallback fonts - Speed Kit, acessado em fevereiro 16, 2026, https://www.speedkit.com/blog/reducing-layout-shift-with-custom-fallback-fonts
10. Optimizing web fonts - Vercel, acessado em fevereiro 16, 2026, https://vercel.com/blog/optimizing-web-fonts
11. Best Bento Grid Design Examples [2026] - Mockuuups Studio, acessado em fevereiro 16, 2026, https://mockuuups.studio/blog/post/best-bento-grid-design-examples/
12. Apple's Bento Grid Secret: How a Lunchbox Layout Sells Premium Tech - Medium,

acessado em fevereiro 16, 2026, https://medium.com/@jefyjery10/apples-bento-grid-secret-how-a-lunchbox-layout-sells-premium-tech-7c118ce898aa
13. Bento Grid Design: How to Create Modern Modular Layouts in 2026 - Landdding, acessado em fevereiro 16, 2026, https://landdding.com/blog/blog-bento-grid-design-guide
14. Brand New Layouts with CSS Subgrid - Josh W. Comeau, acessado em fevereiro 16, 2026, https://www.joshwcomeau.com/css/subgrid/
15. Fluid Typography with Font Clamp - Elementor Course 012 - YouTube, acessado em fevereiro 16, 2026, https://www.youtube.com/watch?v=2ncDXRX1iMU
16. How to build a Responsive Bento Grid with Tailwind CSS (No Masonry.js) - DEV Community, acessado em fevereiro 16, 2026, https://dev.to/velox-web/how-to-build-a-responsive-bento-grid-with-tailwind-css-no-masonryjs-3f2c
17. SwiftUI Spring Animation Cheat Sheet for Developers - GitHub, acessado em fevereiro 16, 2026, https://github.com/GetStream/swiftui-spring-animations
18. Spring animations - animations.dev, acessado em fevereiro 16, 2026, https://animations.dev/learn/animation-theory/spring-animations
19. A Friendly Introduction to Spring Physics Animation in JavaScript ..., acessado em fevereiro 16, 2026, https://www.joshwcomeau.com/animation/a-friendly-introduction-to-spring-physics/
20. Learning SwiftUI Spring Animations: The Basics and Beyond | by Amos Gyamfi | Medium, acessado em fevereiro 16, 2026, https://medium.com/@amosgyamfi/learning-swiftui-spring-animations-the-basics-and-beyond-4fb032212487
21. 2 Ways to Make a Magnetic Buttons using React, GSAP, Framer ..., acessado em fevereiro 16, 2026, https://blog.olivierlarose.com/tutorials/magnetic-button
22. Top 5 React Animation Libraries: Bring Life to Your Web Applications - DEV Community, acessado em fevereiro 16, 2026, https://dev.to/riteshkokam/top-5-react-animation-libraries-bring-life-to-your-web-applications-2hm8
23. Beyond Eye Candy: Top 7 React Animation Libraries for Real-World Apps in 2026, acessado em fevereiro 16, 2026, https://www.syncfusion.com/blogs/post/top-react-animation-libraries
24. The Web Animation Performance Tier List - Motion Magazine, acessado em fevereiro 16, 2026, https://motion.dev/magazine/web-animation-performance-tier-list
25. Exploring Motion One from Framer Motion - LogRocket Blog, acessado em fevereiro 16, 2026, https://blog.logrocket.com/exploring-motion-one-framer-motion/
26. Comparing the best React animation libraries for 2026 - LogRocket Blog, acessado em fevereiro 16, 2026, https://blog.logrocket.com/best-react-animation-libraries/
27. Infinite Marquee Animation using Modern CSS – Frontend Masters Blog, acessado

em fevereiro 16, 2026, https://frontendmasters.com/blog/infinite-marquee-animation-using-modern-css/
28. Next-Level CSS in 2026 — No JavaScript Needed? Playing with siblings. - YouTube, acessado em fevereiro 16, 2026, https://www.youtube.com/watch?v=q08uwV2tP3Y
29. Responsive Infinite Logo Marquee - CSS Tip, acessado em fevereiro 16, 2026, https://css-tip.com/logo-marquee/
30. You Can't Find a Better Infinite Marquee - YouTube, acessado em fevereiro 16, 2026, https://www.youtube.com/watch?v=ZMCNin2VjxU
31. Infinite horizontal scroll marquee - pause on hover - resume in same direction - GSAP, acessado em fevereiro 16, 2026, https://gsap.com/community/forums/topic/39535-infinite-horizontal-scroll-marquee-pause-on-hover-resume-in-same-direction/
32. How to Make an Animated Cursor using React and GSAP, acessado em fevereiro 16, 2026, https://blog.olivierlarose.com/tutorials/blend-mode-cursor
33. Design Engineering at Vercel: What we do and how we do it - Vercel, acessado em fevereiro 16, 2026, https://vercel.com/blog/design-engineering-at-vercel
34. Connect: behind the front-end experience - Stripe, acessado em fevereiro 16, 2026, https://stripe.com/blog/connect-front-end-experience
35. Springs and Bounces in Native CSS The magic of the linear() timing function - Josh Comeau, acessado em fevereiro 16, 2026, https://www.joshwcomeau.com/animation/linear-timing-function/
36. Steal These 2026 Web Design Trends - YouTube, acessado em fevereiro 16, 2026, https://www.youtube.com/watch?v=waHuVF3XuMA