ChatGPT

OKLCH-based Design Token and Color System for a Financial App

Designing an accessible, themeable color system using OKLAB/OKLCH ensures consistent contrast and a harmonious palette across light, dark, and high-contrast modes. Below we outline a comprehensive approach, covering design tokens, color math, accessibility (APCA contrast), multi-mode support, integration with Tailwind/Next.js, and user customization.

Color Design Tokens and Semantic Palette

Semantic Tokens: Define color tokens by semantic purpose rather than by raw color name [1]. For example, use tokens like:

- **background (bg):** overall app background.
- **surface:** surfaces/cards UI elements.
- **ink (text):** primary body text color.
- **ink strong:** high-emphasis text (headings).
- **accent1 / accent2:** primary and secondary brand accents.
- **success / warning / danger:** status colors for feedback.

Each token represents a role (e.g. text, surface, accent) rather than a specific hue [1]. This semantic naming makes the system easier to maintain and theme (e.g. "accent" can change per brand or theme without code changes).

Token Format (JSON/YAML): Store tokens in a technology-agnostic JSON/YAML structure, separate from implementation. This allows generating platform-specific outputs (CSS, TS, etc.) via tools like Style Dictionary [2]. For example, a JSON snippet for light theme tokens could be:

```
{
  "color": {
    "bg":        { "value": "oklch(98% 0.00 240)" },
    "surface":   { "value": "oklch(95% 0.01 240)" },
    "surfaceAlt":{ "value": "oklch(92% 0.01 240)" },
    "text":      { "value": "oklch(22% 0.02 240)" },
    "textMuted": { "value": "oklch(40% 0.01 240)" },
    "border":    { "value": "oklch(82% 0.01 240)" },

    "accent1": {
      "100": { "value": "oklch(90% 0.05 250)" },
      "70":  { "value": "oklch(70% 0.09 250)" },
      "60":  { "value": "oklch(60% 0.12 250)" },
```

```
      "50":  { "value": "oklch(50% 0.14 250)" },
      "40":  { "value": "oklch(40% 0.12 250)" }
    },
    "accent2": {
      "60":  { "value": "oklch(62% 0.15 130)" }
      /* ...additional steps if needed... */
    },

    "success":   { "value": "oklch(60% 0.12 145)" },
    "warning":   { "value": "oklch(65% 0.18 80)" },
    "danger":    { "value": "oklch(58% 0.18 30)" }
  }
}
```

Illustrative snippet: The above JSON defines semantic tokens with OKLCH values. For instance, `bg` is a near-white background (`L=98%`), `text` is dark text (`L=22%`), etc. The **accent1** token includes a **ramp** of values (100, 70, 60, 50, 40) corresponding to different lightness levels of the primary brand color. By convention, lower numbers are lighter and higher numbers darker [3]. The **accent2** token could represent a secondary brand color. Status tokens (success, warning, danger) have distinct hues (greenish for success, yellow for warning, red for danger). In practice, you would generate similar JSON for dark and high-contrast themes (or include mode-specific sub-keys) to capture different values per mode.

This token file can be processed to output Tailwind-compatible variables or JSON for consumption in the app. For example, Style Dictionary can export such tokens to CSS custom properties (e.g., `--color-bg`, `--color-text`, `--accent1-60`) and JSON/TS files [2] [4]. The tokens remain the single source of truth, separate from implementation, ensuring consistency across platforms.

Programmatic Color Ramps with OKLCH

Anchor OKLCH Values: Each semantic color has an **anchor** defined in OKLCH, which serves as the mid or base value for that token's ramp. For example, the primary accent might have an anchor at `oklch(0.60 0.12 250)` (L=0.60, moderate chroma, hue 250) [5] – this could be the base color for accent backgrounds (like buttons). Neutrals (grays) use very low chroma anchors (near 0) to stay grayish [6].

Generating Ramps (Neutral & Accent): From each anchor, create a **ramp** by systematically varying Lightness (L) and Chroma (C) while keeping Hue (h) constant [7] [8]. In OKLCH, equal steps in L correspond to equal perceived brightness changes [7], so adjusting L gives a smooth progression from light to dark. For a **neutral ramp** (grays), use a near-zero chroma (e.g., C=0.01–0.03) and vary L from very light to very dark. For an **accent ramp**, use the anchor's hue and adjust L up and down to get lighter and darker variants, possibly also modulating chroma slightly to maintain visual balance (often, lighter tints need lower chroma to avoid neon glow [9]).

Example: If the base accent1 is `oklch(60% 0.12 250)`, one could generate a 5-step ramp by choosing target L values, say 90% (very light), 70%, 60% (base), 50%, 40% (very dark). Starting from the base, lighter steps (70, 90) would have higher L and slightly lower chroma, and darker steps (50, 40) lower L and possibly slightly reduced chroma near the darkest to avoid gamut issues [9]. This yields a consistent "accent1"

palette from lightest (for highlights or backgrounds) to darkest (for text on accent or hover states). Ethan Gardner provides a similar example ramp for a blue color where L ranges ~97% to 65% as the shade gets darker [3].

For **sequential ramps** (e.g., heatmaps), one can algorithmically interpolate between a light desaturated color and a dark saturated color while keeping hue constant [10]. For instance, to create a blue data visualization ramp with *n* steps, pick a lightness range (L_high=95% for lightest, L_low=40% for darkest) and a chroma range (C_low=0.04 for the lightest tint, C_high=0.15 for darkest) [10]. Then interpolate intermediate L and C values linearly for the remaining steps [10]. The hue stays fixed (e.g., the hue for blue). The result is a smooth progression where the lightest color is a near-white with a hint of blue, and the darkest is a rich dark blue [11].

Month-by-Month Hue Variations: For charts or heatmaps that need distinct colors per month (12 variants), utilize a **hue rotation** strategy. Starting from a base accent hue, generate 12 related hues by shifting the hue by an equal increment each month (≈30° steps for 12 divisions of the color wheel) [12]. Crucially, keep Lightness and Chroma consistent across these monthly variants so they appear harmoniously related [13]. For example, if the base is `oklch(60% 0.15 210)` (a certain blue), January could use hue 210°, February 240°, March 270°, etc., all with L=60%, C=0.15. This yields a categorical palette of 12 colors that feel uniform in brightness and intensity [13] – ideal for differentiating months in data visualizations while maintaining a cohesive look. (If more differentiation is needed, you can pair hue shifts with slight lightness tweaks, but keep within contrast guidelines.)

Color Harmony and Guardrails: By designing ramps in OKLCH, we ensure colors maintain consistent perceived lightness and avoid unexpected jumps between shades [7]. We also set guardrails on L and C for each semantic group [14] to enforce harmony and accessibility. For example, define that *text colors* in light mode must stay below L=30% (to always be dark enough on a light background), and above L=80% in dark mode (to be light enough on dark) [14]. Similarly, *accent background* colors might target an L around 55–65% so that they work in both light and dark modes with minor adjustments [14]. Limit chroma for UI neutrals (borders, surfaces) to very low values (C ~0.01–0.03) to avoid any unintended color cast or "halo" on neutral elements [6]. For vibrant accents, set a reasonable chroma ceiling (e.g. $C \leq 0.15$ in sRGB) to avoid clipping on standard displays [9]. Documenting these ranges for each token (role) ensures that when new theme variants or brands are introduced, they stay within safe limits [14].

Accessibility: APCA Contrast Enforcement

Accessibility is paramount. We use the **Advanced Perceptual Contrast Algorithm (APCA)** to ensure all text and key UI elements meet contrast targets closer to human vision than legacy WCAG ratios [15] [16]. APCA outputs a contrast *score* (Lc value) roughly from 0 to 105 (higher is more contrast) [17]. Our targets are:

- **Body text:** Lc $\geq$ 60 (sufficient contrast for regular text) [18]. This roughly corresponds to a WCAG 2.0 ~4.5:1 contrast, but APCA's 60 is tuned for at least ~16px bold or 24px regular text [18].
- **Large or bold text:** Lc $\geq$ 45 (for headings, large labels, or bold text) [19]. This is a slightly lower contrast requirement permitted for larger text since it's inherently more legible [18].

These thresholds align with emerging WCAG 3.0 guidance, where *Lc 60* is the recommended minimum for "text you want people to read" and *Lc 45* is minimum for large/heavy text [18]. (For small body text at 14–

18px, even higher values like 75 or 90 are preferred for maximum readability [16] [20], so we aim for 60 as a minimum and higher when possible.)

Applying Contrast Checks: Each token is defined alongside its intended usage context (foreground/background pairing), and we verify its APCA contrast. For example:

- **Text on Surface:** `color.text` (L≈22%) on `color.bg` (L≈98%) in light mode should achieve Lc ≈ 75+, exceeding the 60 target for body text (dark text on light background typically yields a high positive Lc) [21] [16]. In dark mode, `color.text` (L≈93%) on dark background (L≈13%) similarly should exceed Lc 60 (white-ish text on black yields Lc ~ -78; the sign is negative because APCA inverts sign for light-on-dark, but magnitude ~78 > 60) [15] [22].
- **Muted Text on Surface:** `color.textMuted` might be lighter gray (L≈40% on white bg) for less emphasis. We ensure it still hits at least Lc 60 for legibility of secondary text [23]. If not, we darken it or adjust chroma (often reducing chroma can improve contrast perceptually [24]).
- **Text on Accent:** For buttons or accent components, ensure the *accent text* color (whether it's always white or dynamic) has Lc ≥ 60 on the accent background. E.g., if accent1-60 (L~60%) is used as a button background, the text might need to be near black (L~20%) to reach Lc 60+, or if using white text on a 60% L colored background, that likely won't meet Lc 60 (white on mid-light color is low contrast). In practice, we might designate a special `color.accentText` token (for text on colored buttons) that adjusts depending on the theme. For instance, use a very dark variant of the accent hue, or simply black/white as appropriate, to ensure contrast. (Our tokens example includes `--color-accent-text: oklch(0.18 0.03 250)` as a dark text for accent backgrounds [25].)

Contrast Matrix (CSV): To systematically validate contrast, we generate a contrast matrix for each theme. This is a CSV (or table) listing key foreground/background combinations and their APCA scores. For example:

```
Foreground \ Background, Surface(Light), SurfaceAlt(Light), Accent1-60 (Light)
text, 68.5, 60.1, 91.2
textMuted, 45.3, 40.0, 70.5
accentText, 75.0, 70.2, 64.8
...
```

(Values above are illustrative APCA Lc scores; e.g., `text` *on* `Surface` *Lc 68.5, which is above 60 = pass.)* Each theme (Light, Dark, High-Contrast) has its own matrix. In dark mode, for instance, the roles reverse (light text on dark background). We ensure all relevant pairs meet the target: body text vs. primary background, text on surfaces, text on accent backgrounds, etc. This matrix not only documents compliance but also helps designers catch low-contrast pairings at a glance [26].

Automated Contrast Testing: We integrate contrast checks into our pipeline. A TypeScript utility reads the token values, computes APCA contrast for predefined critical pairs, and throws an error if any fall below thresholds [27]. For APCA calculation, we can use the official algorithm (e.g. via the `apca-w3` npm package or a custom implementation referencing the APCA formula). The test might iterate through a list of token pairs, e.g.:

```
const pairsToTest = [
  { fg: tokens.text, bg: tokens.bg, minLc: 60, name: "Body text on
background" },
  { fg: tokens.textMuted, bg: tokens.bg, minLc: 60, name: "Muted text on
background" },
  { fg: tokens.accentText, bg: tokens.accent1['60'], minLc: 60, name: "Button
text on accent" },
  // ... etc
];
pairsToTest.forEach(t => {
  const cr = computeAPCA(t.fg, t.bg);
  if (Math.abs(cr) < t.minLc) {
    throw new Error(`Contrast fail: ${t.name} (APCA ${cr})`);
  }
});
```

This automated suite will flag any token that doesn't meet contrast standards, preventing regressions [27]. We also track traditional WCAG 2.1 contrast ratios in documentation for reference, but APCA is our primary guide for tuning colors [28]. By designing with **Lightness-first** using OKLCH, we inherently create palettes that satisfy contrast in both light and dark modes with minimal adjustment [29] [30].

Additionally, we **throttle chroma for accessibility**: extremely high chroma colors can appear blurry or reduce legibility, especially for small text [24]. Our tokens keep text colors relatively low chroma (near gray for neutrals, or moderate saturation for accent text) to maximize readability [24]. For example, we avoid using a pure neon green for success text; instead we use a duller green for text and reserve the vivid hue for perhaps an icon or background, thereby preserving contrast.

Light, Dark, and High-Contrast Modes

The color system supports **Light mode**, **Dark mode**, and **High-Contrast mode** with appropriate token values for each. We define min/max safe Lightness levels per mode to maintain usability:

- **Light Mode:** Generally uses light backgrounds (high L) and dark text (low L). We avoid absolute white (#FFF) or absolute black (#000) extremes; instead, use slightly offset values to reduce eye strain. For example, background might be L=97% instead of 100%, and body text L=22% instead of 0% [31] [32]. This prevents too stark a contrast while still exceeding APCA targets. Light mode accent colors are chosen at appropriate mid-lightness so they stand out on a light background but aren't too dark to use in dark mode (e.g. accent backgrounds around L 55–65% as mentioned) [14].
- **Dark Mode:** Inverts the scheme – dark backgrounds (low L) and light text (high L). Again, we avoid pure black (#000) backgrounds; e.g., use a dark gray at L ~13% for `surface` [33]. Pure black can be hard on the eyes and also, interestingly, APCA penalizes white text on pure black more than on a slightly lighter dark gray [34]. We set light text tokens at around L 90–95% (near-white) [33], instead of 100%, to likewise avoid glare. **Min Lightness in Dark mode** for text ~80% or higher [14] (to ensure readability), and **Max Lightness in Dark mode** for any UI element maybe ~85–90% (to avoid any element being full white unless absolutely needed for contrast). Dark mode accent colors might be slightly adjusted in lightness compared to light mode to ensure they don't appear too dull or too

bright against dark surfaces. For instance, the light-mode accent base L=60% might be fine on dark too, but if an accent needs to be lighter in dark mode, we provide a token override (e.g., in BoldVanta example, their `--brand-90` was 90% in light, but toned to 85% in dark to avoid being too glaring [33]).

- **High-Contrast Mode:** High-contrast mode (for users who require extra differentiation) uses a specialized palette where the contrast between foreground/background is maximized. This mode might ignore some of the nuanced brand colors in favor of clarity. For example, in high-contrast mode we might use pure black and pure white (or very close) for text and backgrounds to get maximum Lc, and use accent colors only if they meet very high contrast (maybe using bold outlines or underlines for emphasis rather than color alone). We create a separate token set or adjustment layer for high-contrast: e.g., `color.bg` might become `#000` and `color.text` `#FFF` (or vice versa for dark-on-light) in high-contrast theme, and accent colors could be mapped to one of the high-contrast accessible color pairs (like yellow on black, etc., which are commonly high contrast). The JSON tokens can include a `"highContrast"` theme entry, or we maintain high-contrast as an independent theme that overrides the base tokens.

We ensure *all modes* respect the **prefers-contrast media query**. If a user's OS is set to high-contrast, we automatically switch to our high-contrast token set (through a CSS media query or by detecting it in JavaScript and applying a `high-contrast` theme class).

Reduced-Motion Fallbacks: Although colors themselves are static, dynamic color effects (like hover transitions or interactive state changes) should consider users with `prefers-reduced-motion`. We avoid any excessive color flashing or animated color gradients for users who opt out of motion. For instance, if the app has an animated gradient background or a shifting heatmap over time, in reduced-motion mode we provide a static visual alternative (e.g., a static color or slower transitions). All CSS color transitions should be wrapped in `@media (prefers-reduced-motion: no-preference)` so they disable when motion is reduced. Also, ensure that any critical information conveyed through color changes (like an error field glowing) is also conveyed via a non-motion indicator (like a static outline). Our QA process includes verifying the interface at reduced motion settings and high contrast settings [35] to catch any issues. For example, we check that a focus state is still visible without an animation, or that an interactive chart can be interpreted without needing a cycling color animation [35].

Integration with Next.js, React, and Tailwind CSS

Integration of this design token system into a Next.js/React app with Tailwind CSS involves exporting the tokens to usable formats and applying them at runtime for theming:

- **CSS Custom Properties:** One approach is to output the design tokens as CSS variables (e.g., in a `:root` selector for default theme). For example, generate a CSS file (or inject in global styles) containing:

```
:root {
  --color-bg: oklch(98% 0 240);
  --color-surface: oklch(95% 0.01 240);
  --color-text: oklch(22% 0.02 240);
  --color-text-muted: oklch(40% 0.01 240);
```

```
  --color-border: oklch(82% 0.01 240);
  --color-accent1-60: oklch(60% 0.12 250);
  --color-accent1-70: oklch(70% 0.09 250);
  /* ... etc ... */
}
@media (prefers-color-scheme: dark) {
  :root {
    /* override to dark theme values */
    --color-bg: oklch(13% 0 240);
    --color-surface: oklch(18% 0.01 240);
    --color-text: oklch(93% 0.02 240);
    --color-text-muted: oklch(75% 0.01 240);
    --color-border: oklch(28% 0.01 240);
    --color-accent1-60: oklch(65% 0.10 250); /* example tweak for dark */
    /* ... */
  }
}
/* Similarly, a high-contrast mode media query or class could override
variables */
```

This uses the OS `prefers-color-scheme` media query to automatically apply light or dark token values [36]. High-contrast mode can be detected with `@media(prefers-contrast: more)` or by providing a manual toggle that swaps to a `.theme-high-contrast` class on the root, which overrides the variables with high-contrast values.

- **Tailwind Configuration:** Tailwind can consume these tokens in multiple ways. You can leverage Tailwind's theming to map design token values to Tailwind's utility classes:

- The simplest is to use the CSS variables directly in Tailwind's utilities via the JIT engine. For example, you might write classes like `bg-[var(--color-bg)]` or text color classes using `text-[var(--color-text)]`. Tailwind will preserve these as-is. This approach requires minimal config and directly ties to the CSS vars.
- Alternatively, extend Tailwind's theme in `tailwind.config.js` by injecting the token values. For example:

```
// tailwind.config.js
module.exports = {
  theme: {
    colors: {
      bg: 'rgb(var(--color-bg) / <alpha-value>)',
      surface: 'rgb(var(--color-surface) / <alpha-value>)',
      text: 'rgb(var(--color-text) / <alpha-value>)',
      accent1: {
        60: 'rgb(var(--color-accent1-60) / <alpha-value>)',
        70: 'rgb(var(--color-accent1-70) / <alpha-value>)',
        // ...
```

```
      },
      // ...
    }
  },
  darkMode: 'media', // or 'class' if using a manual class toggle
  // ...
}
```

Here we configure Tailwind to use our CSS custom properties for colors. The `/<alpha-value>` part allows Tailwind's opacity utilities to work. This way, `bg-bg` in Tailwind would use the `--color-bg` variable. In dark mode, since the CSS variable is overridden, the same Tailwind class automatically reflects the dark theme color. Tailwind's `darkMode` setting can be `'media'` (to follow OS preference) or `'class'` (to toggle via a `.dark` class on `<html>`). If we want to allow user override of light/dark, `'class'` mode is useful: we'd add or remove a `dark` class on the `<html>` element based on user choice (using Next.js dynamic theming logic, e.g. with a context or a library like `next-themes`).

- **Tailwind v4 `@theme` (if available):** Tailwind's newer versions have an `@theme` directive for design tokens. We could define our tokens in a `:root` using `@theme` which automatically ties them to utilities. For example:

```
@theme {
  --color-bg: 98% 0% 94%; /* example in HSL or percent form for fallback */
  --color-text: ...;
  /* etc */
}
```

This is an advanced feature that essentially integrates tokens at build time.

- **Applying Themes in React:** At runtime, the default will follow OS preferences (thanks to `prefers-color-scheme` in CSS or initial `darkMode: 'media'`). For user overrides, we can implement a theme toggle that adds a `dark` or `high-contrast` class to the `<html>` or `<body>` tag. Our CSS variables and Tailwind classes respond to these. For high-contrast, we might use a class `.theme-hc` to override variables (since not all browsers support `prefers-contrast` yet). We ensure that all components use tokens (via Tailwind classes or CSS vars) for colors, so that switching the theme class or media query flips the entire UI's colors consistently.

- **Charts and Data Visualizations:** We allow charts (e.g., using D3 or Chart.js in React) to reference the same design tokens so they blend with the UI. Instead of hardcoding chart colors, the chart components should pull from the token JSON or CSS variables. For example, a bar chart with multiple categories can use `accent1-60`, `accent1-70`, etc., or the month-by-month palette derived from `accent2` hue rotations. We might expose a small utility that given a number of series or categories, returns an array of token values (for instance, for 12 months, it returns the 12 precomputed OKLCH hues). By doing this, if the theme changes (say user switches to a different accent color or dark mode), the chart colors update too, maintaining readability. In high-contrast mode, charts might switch to a specialized palette (perhaps using patterns or high-contrast colors) if

necessary for accessibility. Chart and heatmap color scales can also be semantic: e.g., a heatmap for "intensity" might use `accent1` ramp from token (light to dark of accent1), whereas a divergent chart might use `success` vs `danger` ramps for positive/negative. All these ensure consistency with the design language.

- **System-wide Overrides:** The system allows an admin or user to apply an override theme globally (for example, a custom corporate theme in a SaaS context). Our architecture supports this by isolating all color definitions in the tokens. An override can supply a new JSON of tokens (perhaps generated or provided via theming UI) and the app can swap to it. Technically, this could mean swapping the CSS variables values (via a different CSS file or injecting a `<style>` with the new :root definitions). Because components use semantic references (e.g., `color.bg`), a wholesale swap is straightforward. If partial overrides are allowed (say only change accent color globally), we can merge the override tokens with the base tokens and apply. Using CSS variables, overriding one variable (like `--color-accent1-60`) is trivial and will propagate everywhere that color is used.

In summary, integration is achieved by **connecting the token source to the UI styles**. The design tokens JSON is the single source — from it we generate CSS variables (for runtime theming) and a Tailwind theme (for utility classes). Next.js can import the JSON at build time for server-side rendering critical styles or use it to feed an inline `<style>` for initial colors (preventing a flash of unthemed content). Because we use OKLCH in CSS, we get consistent results across modern browsers, and can fallback to HEX if needed for older support via a build step [37].

User Customization and Dynamic Palettes

We want to enable end-users (or clients) to customize the app's color theme safely. Key aspects of user customization:

Base Color Selection: Users can pick a **base color** (e.g., their brand or a preferred accent) via a color picker input. We take this color (in hex, RGB, etc.), convert it to OKLCH, and use it as the new base for accent tokens. For instance, if the user picks a color #E36D5A (an orange), we convert to OKLCH (say ~L 0.65, C 0.18, h 30). This becomes the new anchor for `accent1`. Immediately, we regenerate the accent ramp: we keep the same target Lightness values as our default ramp (to maintain contrast), but apply the user's hue (and possibly adjust chroma). Essentially, we **inherit the L structure** from the system and **swap hue/chroma** to the user's selection [30]. This way, all derived tokens still meet contrast because the L values were chosen to be accessible [30]. We might clamp extremely high chroma to our max to ensure the color isn't out of gamut or low contrast [9]. For example, if the user's chosen color had a chroma of 0.3 (very saturated) but our safe limit is 0.15, we tone it down to 0.15 – the resulting color will be slightly less saturated but still recognizably in the same hue family, and it will work with text.

Harmonious Palette Generation: Beyond the main accent, we can automatically generate complementary colors so that the whole UI remains balanced. Techniques include: - **Hue rotation:** Use the base color's hue and create additional accents by rotating the hue by fixed amounts [12]. For example, if the base is a blue (h=210), we might set accent2 to an analogous hue (like 170 or 250) or a complementary (210+180=30, a orange hue). This provides a second accent that still looks coherent. We keep their lightness and chroma aligned to appear as a set [13]. - **Lightness scaling:** If user can customize beyond just accent (say they choose an overall theme brightness), we could scale the neutral palette lighter or darker. For instance, a

user might prefer a “dim” theme even in light mode – we could take all neutral L values and reduce them slightly (within contrast constraints) to produce a dimmer light theme. - **Chroma adjustments:** If a user finds the theme too vibrant or too dull, we could offer a “vibrancy” slider that adjusts chroma of accent tokens within safe bounds. Increasing chroma will make accents more saturated (but we must ensure contrast of text on those accents is re-checked), decreasing chroma moves them closer to grayscale.

All these customizations occur through **algorithmic enforcement of harmony**. When one token changes, related tokens update to preserve overall balance. For example: - If the user changes the base accent hue, we recalc the entire accent ramp (light to dark variants) using that hue. The neutral tokens remain unchanged (so readability of text on background is unaffected), but we might subtly shift neutrals’ hue tint if desired (some systems tint grays slightly toward the brand color). This could be an option: e.g., neutrals have a slight blue cast in the default; if user picks a green brand, we might allow an optional “tint neutrals” toggle to mix a bit of that hue into surfaces/borders for a cohesive feel. If done, it would be a slight chroma introduction (like C 0.02 at most) with the new hue. - If the user adjusts a neutral (say background color), we immediately adjust text to maintain contrast. For instance, suppose we allow users to pick a global background color (light theme) – if they choose an off-white with L=90%, our system would detect that body text at L=22% might drop below APCA 60. The system could respond by darkening the text token until the APCA check passes (or by warning the user that this choice reduces contrast). We prefer automatic adjustment to ensure accessibility: e.g., “As you have chosen a darker background, we have adjusted the text to be slightly lighter (from L=22% to 25%) to maintain contrast.” - If a user picks a very light accent color (close to white), using it as a button background in light mode could fail contrast with white text. Our live APCA check would flag that, and we could automatically switch the button text to dark in that scenario. Essentially, `accentText` token can dynamically choose black or white based on the computed L of the accent background (a common technique, similar to YIQ or L threshold logic, but we use APCA for accuracy). We can smoothly transition between using dark or light text on accent depending on the background’s L (for instance, if accent L > 60% in light mode, use dark text; if accent L < 60%, white text might suffice). - For custom charts: if users can choose custom data colors, we also run those through APCA checks against backgrounds and adjust lightness if needed. For example, a user might pick a color for a pie chart slice that is too light and not distinguishable on a white background; we could automatically deepen it or add an outline in high-contrast mode.

Live APCA Feedback: The customization UI should show live contrast indicators. As the user tweaks a color, we compute APCA values for key elements (like that color as a background with text) and display a pass/fail or a numeric score. This guides the user to make accessible choices. We may restrict the picker’s range to avoid entirely inaccessible picks (for instance, if they try to choose pure yellow at full lightness for text, we can warn or disallow because no contrast). This way, the user is empowered to theme, but **cannot accidentally violate contrast standards easily**.

Harmony Enforcement Algorithm: When a token is changed, we propagate changes to related tokens: - Use relative relationships rather than absolute. For example, we maintain a ratio or difference in Lightness between `surface` and `border` or between `accent-bg` and `accent-hover`. Instead of fixed values, they can be defined as functions: e.g., `accent-hover = lighten(accent-base, +5 L in OKLCH)` for light mode (or darken for dark mode) [38]. By storing these relationships (maybe in code or in design documentation), any base change still yields a consistent outcome (like hover always slightly lighter or darker appropriately) [38]. - If one accent changes, secondary accent might optionally shift by the same hue offset relative to it as before (ensuring their contrast remains similar). Or if the user specifically sets both accents, we respect that. We provide sensible defaults but let advanced users fine-tune. - Ensure **contrast**

invariants: For any user change, run through the contrast test suite again. If something fails, either automatically correct it or highlight it to the user. For automatic correction, an approach is to adjust Lightness of either the foreground or background until the threshold is met. Because we're in OKLCH, we can do this by small increments of L or reducing C. For example, if user picks a very pale yellow for text (which might be hard to read on white), our algorithm could darken that yellow until Lc 60 is reached, and show the adjusted result. In practice, we might simply not allow setting a text color above a certain Lightness or an accent background above a certain Lightness if the text on it is white. - Provide visual feedback for out-of-harmony situations. For instance, if a user tries to set two accents that clash (e.g., wildly different saturations or clashing hues), we could suggest making them analogous or adjusting one's chroma to better pair with the other. This can be based on color harmony rules (complementary, triadic, etc.). While full color-theory enforcement is complex, simple checks like "if new accent2 hue is very close to accent1 hue but not the same, maybe the user intended a distinct color – consider picking a more contrasting hue" can be implemented.

Overall, the system takes the **user's input color as a seed** and **derives the full token set from it** using our predefined OKLCH transformations. This ensures that a single change results in a *harmonious, accessible palette* rather than a jumble of inconsistent colors. The use of OKLCH makes these transformations intuitive (hue rotations, lightness adjustments) while preserving perceptual uniformity [13] [39]. By leveraging APCA during customization, we guarantee that even custom themes uphold readability standards, fulfilling both user preference and accessibility.

Deliverables

- **Design Token JSON Schema:** A structured JSON schema defining all color tokens, their semantic names, and values per theme. This includes neutral tokens (background, surface, text, etc.), accent tokens (with ramps), and state tokens (success, warning, danger). The schema is organized for easy export (e.g., with Style Dictionary) to various formats. *We provided a sample JSON snippet illustrating this structure, with OKLCH values for a light theme.* The full schema would enumerate tokens for light, dark, high-contrast, and could be extended with brand-specific overrides. Each token entry includes metadata like its purpose and perhaps comments on contrast requirements.

- **Sample OKLCH Color Ramps:** Examples of generated color ramps for neutral and accent colors. *For instance, a neutral gray ramp from* `bg` *(almost white) down to* `ink` *(almost black) in equal perceptual lightness steps, and an accent1 ramp from a light tint (accent1-100) to a base (accent1-60) to a dark shade (accent1-40). We described how accent1 ramp is derived from a single OKLCH anchor by adjusting L and C while holding hue constant* [11]. The deliverable would include actual values for these ramps, possibly visualized as a set of swatches, to demonstrate smooth progression. We also outline a month-by-month accent variation (12-hue palette) as a special ramp for data visualization.

- **Contrast Matrix (CSV):** A CSV file (or spreadsheet) listing APCA contrast values for key foreground/background combinations in each theme. *We gave an example snippet of how this might look in CSV form, with tokens as rows/columns and APCA scores* [23]. The full deliverable would include all relevant pairs (each text color against each background it might appear on, icons on backgrounds, etc.). This matrix would be provided per theme (or labeled accordingly) and can be used by QA or designers to verify all combinations. It doubles as documentation for which combinations are safe and which are not recommended.

- **Theming & Integration Guide:** Documentation describing how to use these tokens in the target tech stack. This guide covers:

  - Setting up CSS variables or Tailwind config with the tokens.
  - Applying light, dark, high-contrast themes (via media queries or classes) in Next.js.
  - Using the tokens in React components (examples of both using Tailwind utilities and CSS-in-JS or style objects referencing the tokens).
  - How to update tokens or add a new theme (e.g., if adding a new branded theme, how to supply new hue values as per the guide).
  - Guidelines for charts using the token system (with examples of pulling token values for a chart library).

  - Best practices for maintaining the token source of truth and syncing it with the app (perhaps mentioning an npm package or style dictionary build step that the developers should run when tokens change). This guide ensures developers and designers know how to implement and extend the system correctly.

- **TypeScript Utilities:** A set of utility functions and tests in TypeScript:

  - **Color Ramp Generator:** A function (or module) to generate color ramps in OKLCH. For example, `generateRamp(baseColor: OKLCH, rampPoints: number[]): OKLCH[]` that takes a base color and outputs an array of OKLCH values for each point (like 40, 50, 60, etc.). It encapsulates the logic of adjusting L and C as per our ramp strategy (could use interpolation or predefined offsets). This can be used to programmatically regenerate ramps when a user selects a new base color.
  - **APCA Contrast Calculator:** A function `calculateAPCA(fg: Color, bg: Color): number` that returns the APCA Lc value. This would implement the APCA algorithm (using either our own implementation or a library) so that we can test contrast dynamically. We might also include a helper `passesContrast(fg, bg, mode: 'body'|'large'): boolean` which checks against the 60 or 45 threshold as appropriate.
  - **Contrast Test Suite:** A script or set of unit tests that iterate over the token set to ensure all required contrast pairs pass. This likely reads a JSON of expected pairs and thresholds and asserts the APCA values meet them [27]. It can output a report or fail the build if any violation is found. This is to be run as part of CI.
  - **Color Conversion and Gamut Checks:** Utility to convert between color formats (especially if user inputs HEX or we need to output HEX for older browsers). Also functions to clamp colors to sRGB gamut if needed (since OKLCH allows colors outside sRGB, we should ensure our chosen values have fallbacks [40]).
  - **User Customization Hooks:** If we provide a live editor in-app, a set of functions to handle user color changes: e.g., `setBaseColor(oklch)` which calls the ramp generator, updates CSS variables, and re-runs contrast checks. Possibly a hook that components can use to subscribe to theme changes (leveraging React context or a Zustand store, etc.).

All deliverables aim to create a robust, future-proof color system. By using OKLCH, we gain predictable, perceptually uniform color behavior across themes [39], and by enforcing APCA contrast, we align with next-generation accessibility standards [16] [18]. The result is a design token system that is accessible (meeting contrast targets), multi-mode (supporting light, dark, high-contrast), themable (easily swapped or extended for new themes), and user-customizable (with algorithms maintaining harmony and contrast automatically).

This ensures the financial app's UI is not only visually appealing but also adaptable to user needs and future design requirements.

Sources:

- Lena Marlowe, *"Designing Luminance-First Color Systems with OKLCH: Tokens, Ramps, and Real-World Pitfalls."* BoldVanta (Jul 14, 2025) – *Guidance on OKLCH-based design tokens, maintaining consistent L (luminance) across themes, and automating contrast checks* [1] [14] [30] [27] .

- Chris Henrick, *"Color experiments with OKLCH."* (Sept 1, 2024) – *Techniques for generating color palettes using OKLCH, including categorical palettes via hue rotation and sequential palettes via lightness/chroma scaling* [12] [10] [8] .

- APCA Contrast Guidelines – Ekaterina Leudarovich, *"APCA: The new color contrast standard for web accessibility."* (Sep 18, 2025) – *Overview of APCA contrast scoring and recommended thresholds (Lc 45 for large text, 60 for body text, etc.)* [18] , and Steve Frenzel's article on APCA (Aug 6, 2023) – *comparison of APCA vs WCAG and list of APCA levels* [16] .

- Ethan Gardner, *"Supporting Color Contrast in Design Systems."* (Jul 19, 2023) – *Discussion on using OKLCH for design tokens and integrating contrast info; includes example JSON of color ramps in OKLCH* [3] [37] .

- Evil Martians, *"OKLCH in CSS: why we moved from RGB and HSL."* (Sep 17, 2025) – *Benefits of OKLCH for design systems (predictable lightness for better a11y, palette generation, etc.)* [39] .

---

[1] [5] [6] [7] [9] [14] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [35] [36] [38] Designing Luminance-First Color Systems with OKLCH: Tokens, Ramps, and Real-World Pitfalls | BoldVanta
https://www.boldvanta.com/design/designing-luminance-cefirst-color-systems-with-oklch-tokens-ramps-and-real-ceworld-pitfalls.html

[2] [3] [4] [37] [40] Supporting Color Contrast in Design Systems | Ethan Gardner
https://www.ethangardner.com/posts/supporting-color-contrast-accessibility/

[8] [10] [11] [12] [13] Color experiments with OKLCH – Chris Henrick
https://clhenrick.io/blog/color-experiments-with-oklch/

[15] [16] [21] [22] [34] How the Washington Post design system made me learn about Perceptual Contrast (APCA) - Steve Frenzel
https://www.stevefrenzel.dev/posts/learning-about-perceptual-contrast/

[17] [18] [19] [20] APCA: The new color contrast standard for web accessibility | by Ekaterina Leudarovich | Bootcamp | Sep, 2025 | Medium
https://medium.com/design-bootcamp/apca-the-new-color-contrast-standard-for-web-accessibility-f634511a3462

[39] OKLCH in CSS: why we moved from RGB and HSL—Martian Chronicles, Evil Martians' team blog
https://evilmartians.com/chronicles/oklch-in-css-why-quit-rgb-hsl