ChatGPT

Financial Calendar Heatmap: Technical & UX Specification

Color Logic (OKLCH Hue Shifts & Intensity Mapping)

Monthly Hue Shift with OKLCH

Each month's calendar receives a distinct base color by shifting the hue in the OKLCH color space. We use a fixed perceptual lightness (L) and chroma (C) for consistency, changing only the hue (H) by about ±20° each month. OKLCH is chosen because it preserves perceptual contrast when adjusting hue or chroma [1]. This means sliding the hue yields consistent visual results without custom tweaking [2]. For example, if January's base hue is `H0`, then:

- February's base hue = `H0 + 20°` (wrapping around 360° as needed)
- March's base hue = `H0 + 40°`, and so on (using a +20° step per month).

All base hues share the same lightness and chroma. For instance, we might use **L = 90%** and **C = 0.15** for light mode (a pastel saturation), ensuring a gentle tint. In dark mode, we use a darker lightness baseline (e.g. **L = 30%**, C = 0.15) to produce a dimmer colored tile suitable for a dark background. By keeping L and C fixed across months, the contrast for text remains predictable as hue changes [1].

OKLCH Color Formula: Each month's base color can be defined as:

```
const baseHue0 = 210; // starting hue for month 0 (e.g. 210° for a blue tone)
const hueStep = 20;   // 20° hue shift per month
const monthHue = (baseHue0 + monthIndex * hueStep) % 360;

// Fixed lightness/chroma for base intensity (full-color days)
const L_lightMode = 90;    // light mode lightness (percent)
const L_darkMode  = 30;    // dark mode lightness (percent)
const C_base      = 0.15;  // chroma for full intensity

// Example CSS color strings:
const baseColorLight  = `oklch(${L_lightMode}% ${C_base} ${monthHue})`;
const baseColorDark   = `oklch(${L_darkMode}% ${C_base} ${monthHue})`;
// Tailwind can use these via CSS variables or inline style
```

This yields a cohesive palette where each month is a distinct hue, but all have equal perceived brightness/saturation. Designers can start with any `baseHue0` – for example, use the user's brand color's hue for the first month – and subsequent months rotate hue by +20° sequentially. (The "±20°" hint implies you can alternate direction or adjust by 20° in a pleasing sequence if needed, but a simple +20° increment works).

OKLCH ensures no unintended hue shifts when adjusting chroma/lightness, making the monthly color transitions smooth [1].

Daily Intensity Buckets (Quantile-Based Color Mapping)

Individual days are colored by adjusting the intensity of the month's base color according to the user's spending on that day. We define a set of **intensity buckets** based on spend quantiles (see "Quantile Bucketing" below). Each bucket corresponds to a different shade of the base color – higher spending → more intense color.

To maintain accessibility, we vary primarily the chroma (saturation) while keeping lightness fixed per theme. This way, every day cell in a given theme has the same brightness, and text contrast against each cell is consistent. For a day in bucket `k` (where `k=0` is lowest spend and `k_max` is highest):

- **Light theme:** Use the month's hue with lightness ~90% throughout. Set chroma proportionally to the bucket level. For example, if `k_max = 10` (10 buckets) and `k = 10` is the highest bucket, use full chroma `C_base` (the base saturation). If `k = 0` (very low spend), use a very low chroma (near grayscale). We can compute `C = C_base * (k / k_max)` for a linear scale. This makes low-spend days very faint pastel and high-spend days fully saturated. Lightness stays at 90% for all, so black text stays readable.

- **Dark theme:** Use the month's hue with lightness ~30% for all intensities. High-spend days use full chroma (0.15 in our example) at L=30%, appearing as a richly colored dark tile. Low-spend days reduce chroma toward 0 (approaching a neutral dark gray at L=30%). Light text (e.g. white) remains legible on all, since brightness is constant.

OKLCH Intensity Formula: If `fraction = k / k_max` (0 to 1 intensity fraction):

```
// For a given day with bucket k in a given month:
const frac = k / k_max;
const L = (theme === 'dark' ? L_darkMode : L_lightMode);  // fixed per theme
const C = C_base * frac;  // scaled chroma
const H = monthHue;       // from monthly hue logic above
const dayColor = `oklch(${L}% ${C.toFixed(3)} ${H})`;
```

For example, suppose we have 5 buckets (k_max=4). In light mode, a day in the lowest bucket (k=0) might be `oklch(90% 0.0 230°)` (nearly white with a tiny tint of that month's hue), whereas a top-bucket day (k=4) would be `oklch(90% 0.15 230°)` (noticeably colored). In dark mode, the lowest bucket could be `oklch(30% 0.0 230°)` (almost black) vs highest `oklch(30% 0.15 230°)`. This approach ensures color intensity reflects spending, yet all text (dates or labels) remain on a uniform light/dark background for contrast consistency.

Adjusting Intensity Steps: The mapping need not be strictly linear; designers might define a non-linear scale (e.g. smaller chroma jumps for lower buckets and a big jump for the top 90–99th percentile bucket to emphasize outliers). The key is that each bucket has an associated chroma level. Chroma should range from

~0 (no intensity) up to the chosen `C_base`. Lightness could also be tweaked slightly across the range if needed for visual effect, but **ensure it stays within safe bounds to maintain text contrast** (see Accessibility below).

Accessibility (Contrast, Themes & ARIA Support)

Ensuring APCA Contrast ≥ 60 (Light & Dark Mode)

All text and labels over the heatmap must meet *Accessible Perceptual Contrast Algorithm* (APCA) contrast Lc ≥ 60, in both light and dark themes. An APCA value of 60 is the recommended minimum for content text that users need to read [3]. To achieve this:

- **Text Color Selection:** Use black or very dark text on light-colored cells, and white or very light text on dark-colored cells. We dynamically choose a text color that maximizes contrast with the cell background. In practice, for each cell we can compute contrast of white vs the cell color and of black vs the cell color using the APCA formula, and pick the option with the higher |Lc| value (absolute Lc) [4]. For our design, since we fix lightness per theme, this simplifies to always using a standard dark text (e.g. `text-gray-900`) in light mode and a standard light text (e.g. `text-gray-100` or `white`) in dark mode. This yields consistent contrast across all intensities.

- **Theming with Safe Lightness:** We deliberately set the base lightness values to ensure contrast. For light mode, L≈90% for backgrounds is bright enough that black text typically yields Lc > 60 (for reference, black on a 90% L background has high contrast under APCA [3]). We avoid making any day cell lighter than ~94% or darker than ~85% L in light mode – this keeps contrast in a safe range (darker background would reduce black-text contrast; lighter is fine as black-on-white is maximal). In dark mode, we choose L≈30% for cells, which is dark enough that white text has a strong contrast (white on 30% L easily exceeds |Lc| 60). We avoid backgrounds lighter than ~40% L in dark mode, to keep white text above the threshold. If we ever needed an intensity beyond those bounds, we'd adjust the text color or not use those extremes.

- **Contrast Testing:** Incorporate an APCA contrast check in development. For any new color choice (e.g. if adjusting L or C values), test the Lc value with tools or a library. The design should **guarantee Lc ≥ 60** for the smallest text used on the heatmap (likely the day number or a two-letter day abbreviation). In practice, our fixed-L approach means all cells in light theme have identical luminance, so if one cell's text passes APCA, all do. The same holds for dark theme. This is a benefit of OKLCH's predictable contrast handling [1].

- **CSS Implementation:** Using Tailwind, we can utilize the `dark:` variant to swap text colors in dark mode. For example, a day cell element could have classes: `text-gray-900 dark:text-gray-50` to use near-black text in light theme and near-white in dark theme. The background color itself can be an inline style (computed as above) or derived from CSS variables. We ensure these text classes are applied to all interactive elements (day cells, tooltips, etc.). We also consider font weight and size per APCA guidance (e.g. if text is very small, aiming for even higher contrast like Lc 75 might be ideal [5], but given our day cells likely use ~12-14px font for numbers, Lc 60 is acceptable).

Screen Reader Accessibility (ARIA) and Keyboard Navigation

The calendar heatmap must be navigable and understandable for users of assistive technology:

- **Semantic Grid Structure:** Use semantic HTML and ARIA roles to convey the calendar structure. Each month section can be a `<section>` or `<div>` with an accessible name like "January 2024". The grid of days can be a semantic grid or table. We can use a `<table>` with one row per week and seven columns (Sunday–Saturday) – this inherently provides a tabular structure that some screen readers interpret well. Alternatively, use ARIA grid roles: e.g. a container with `role="grid"`, each week row with `role="row"`, and each day cell a `role="gridcell"`. In either case, include proper labels.

- **Labels for Days:** Each day cell should have an accessible label that at least announces the date and the spending level. We do **not** rely solely on color for conveying information, so we provide text or an `aria-label`. For example:

  ```
  <button
    className="day-cell"
    aria-label="Sunday, January 8, 2024 – Spent $45 (High spending)"
    >
    8
  </button>
  ```

  In a pure heatmap (like GitHub contributions graph), the day number might not be visible, but here we can display the date number or simply rely on position. We choose to show the day number in the cell for quick reference, and use a visually hidden description if needed for additional info. Screen readers will read the full aria-label (which includes day of week, date, and spending category). The day-of-week headers (Sun, Mon, etc.) should use `<abbr title="Sunday">Sun</abbr>` so screen readers announce the full name [6].

- **Keyboard Navigation:** We follow WAI-ARIA best practices for calendar grids. Only one day in the grid should be focusable at a time (to avoid tabbing through every single cell) [7] [8]. We implement **arrow-key navigation** within the grid: when a day cell has focus, the **Right Arrow** moves to the next day, **Left Arrow** to previous day, **Down Arrow** moves focus to the same weekday in the next week, and **Up Arrow** to the previous week [8]. Home/End can jump to start or end of the current week, and Ctrl+Home/End (or PageUp/PageDown) might jump to the previous or next month (if we implement those bindings). We ensure that when arrowing past the end of a month, focus wraps to the next month (which may trigger loading more months if needed – see infinite scroll). Tabbing away from the calendar grid should move focus out of the grid container (and conversely, a single Tab into the grid should land on the *currently selected* or focused date).

- **Focus Indicators & States:** Each interactive day cell (we use `<button>` elements for days so they are naturally focusable and clickable) should have a visible focus ring. Tailwind's `focus:outline` utilities or custom styles (e.g. a 2px border) can highlight the focused cell. We also provide a hover style for mouse users (e.g. slightly brighter border on hover) and an ARIA `aria-pressed` or

`aria-selected` state if a day is active/selected (for instance, if clicking opens a details panel, we might mark it as selected).

- **Announcements and Live Regions:** If the user navigates between months (e.g. by scrolling or keyboard shortcuts), announce the new month/year. For example, the month label (like "February 2024") can be given `role="heading"` and `aria-live="polite"` so that when it enters view or focus, screen readers announce it. Alternatively, when the focused day moves from one month to the next via arrow keys, our script can detect the month change and programmatically announce the new month (using `aria-live` region).

- **Long-Press Alternative:** The long-press interaction (to annotate or edit budget) should be made accessible for keyboard and screen reader users. Since “long-press” is not a standard keyboard action, we can offer an alternate UI: for instance, when a day has focus, pressing a specific key (like `Enter` vs `Space` difference, or a context menu key, or a secondary button) could open the annotation dialog. We might also implement a right-click (context menu) as an equivalent to long-press on desktop. All interactive features (view details, add note) should be reachable via keyboard (e.g. after focusing a day, pressing Enter could open details, and pressing a custom shortcut or an on-screen “Edit” button could annotate).

In summary, **use ARIA patterns of an interactive calendar** to ensure that blind or low-vision users can navigate day by day and hear the necessary info. The design will be tested with screen readers to confirm that date, spending, and any selection state are announced clearly.

Quantile Bucketing (Per-User Spending Distribution)

To map daily spending to color intensity, we bucket each day’s spending based on quantiles of the user’s historical spending distribution. Using percentiles (p10, p20, … up to p90, p99) allows the color scale to adapt to each user’s habits, rather than fixed dollar ranges.

Calculating Per-User Spend Quantiles (p10–p99)

For each user, gather all their daily total spending values over the relevant time window (e.g. the past 5 years or whatever data is available). Compute the **percentile values** at 10%, 20%, … 90%, and 99%. These serve as threshold cutoffs for buckets:

- **p10** = value at the 10th percentile (10% of days have spending $\leq$ this).
- **p20** = 20th percentile value, …
- **p90** = 90th percentile,
- **p99** = 99th percentile (a high spend value that only 1% of days exceed).

We include p99 to capture outliers without letting a single extreme day skew the whole scale. Days beyond p99 (the top 1%) will be lumped in the highest bucket so that the color scale isn’t blown out by an outlier.

Pseudocode for Quantile Computation: (Assuming we have an array of daily totals for the user)

```
function computeQuantiles(dailyTotals: number[]): Record<string, number> {
  dailyTotals.sort((a,b) => a - b);
  const n = dailyTotals.length;
  const percentiles = [10,20,30,40,50,60,70,80,90,99];
  const result: Record<string, number> = {};
  for (let p of percentiles) {
    const idx = Math.floor((p/100) * (n - 1));
    result[`p${p}`] = dailyTotals[idx];
  }
  return result;
}
```

This simple approach picks the value at the percentile position (no interpolation). In production, a more precise calculation (linear interpolate between nearest ranks) could be used (e.g. SQL's `PERCENTILE_CONT` or a stats library) to handle cases where the percentile falls between data points.

We will run this either in the backend (during nightly batch in SQL or application code) for each user. We typically compute two sets of quantiles if supporting both historical and rolling windows:

- **Historical quantiles:** computed over all available data (or a very long period, e.g. 5+ years).
- **Rolling quantiles (e.g. 12-month rolling):** computed over a recent window (like the last 365 days). This can make the heatmap reflect more recent spending patterns, which is useful if the user's spending has changed over time.

Assigning Daily Buckets from Quantiles

Once we have the threshold values, we assign each day's spending to a bucket:

- If a day's total spend $\leq$ p10, it falls in **Bucket 1** (very low spend).
- If >p10 and $\leq$ p20, Bucket 2.
- ...
- p90 and $\leq$ p99, Bucket 10 (high spend).
- p99, also Bucket 10 (we treat anything above the 99th percentile as part of the highest bucket).

This yields 10 buckets (1–10) in this scheme. We could optionally designate an "11th" bucket for >p99 outliers, but generally grouping them with the 90–99% days is sufficient for color scaling. The distribution of days per bucket will be roughly 10% in each of buckets 1–9, and 1% (or a few days) in bucket 10 if the user had any outliers beyond p99.

Pseudocode for Bucket Assignment:

```
function assignBucket(spend: number, Q: Record<string, number>): number {
  if (spend <= Q.p10) return 1;
  else if (spend <= Q.p20) return 2;
  // ... and so on ...
  else if (spend <= Q.p90) return 9;
  else return 10;
}
```

For robustness, if a user has many days with $0 spend, p10 might be 0. In that case, all $0 days end up in Bucket 1. That's fine – they will all be colored with the faintest shade. If the user has identical spends on many days, multiple percentile thresholds could be the same value; the bucket assignment still works (some buckets might end up empty if quantiles repeat, which isn't harmful).

Historical vs Rolling Window Modes

We will support two modes in the UI: **historical** (all-time) and **rolling** (recent year, for example). The difference is which quantile set is used to map the colors:

- In **historical mode**, the thresholds Q are based on the user's entire spending history. This means the color scale covers their all-time min to near-max. A benefit is consistency and the ability to compare current vs past spending intensity. However, if the user's spending habits changed, the scale might be skewed (e.g. if they spend much more now than years ago, most recent days might all cluster in high buckets).

- In **rolling mode**, the thresholds Q are based on, say, the last 12 months of data. This makes the heatmap more sensitive to recent variations – colors indicate how a day compares to the recent normal. We will likely implement the rolling window as a 1-year window (it could also be user-selectable like 3 months, etc., but 1 year is a good default to capture seasonality).

The data generation will compute both sets of buckets. The frontend can toggle which to display. Technically, we could store two separate bucket values for each day (one for historical, one for rolling). Another approach is storing separate sets and re-rendering based on choice. The **Data Modeling** section below details how we store this.

Note: If the user toggles modes, we should update the aria-labels appropriately (e.g. "High spending" might mean different actual dollar ranges in each mode). It might be useful to convey which mode is active in the screen reader label of the grid (like "Financial heatmap, historical mode").

Data Modeling (Data Structures & Generation)

JSON Schema for Heatmap Data

We define a JSON structure (or database schema) to deliver the heatmap buckets to the frontend. The data essentially consists of daily bucket assignments for each user (and possibly the actual spend values or thresholds for reference). An example JSON response for a user's heatmap data might look like:

```
{
  "userId": 42,
  "mode": "historical",
  "quantiles": {
    "p10": 5.00,
    "p20": 12.50,
    "p30": 18.00,
    "p40": 25.00,
    "p50": 32.00,
    "p60": 45.00,
    "p70": 60.00,
    "p80": 80.00,
    "p90": 120.00,
    "p99": 300.00
  },
  "days": [
    { "date": "2025-01-01", "total": 34.10, "bucket": 6 },
    { "date": "2025-01-02", "total": 0.00,  "bucket": 1 },
    { "date": "2025-01-03", "total": 210.00,"bucket": 10 },
    ...
  ]
}
```

In this example, the user's p90 is $120 and p99 is $300. January 1, 2025 with $34.10 falls into bucket 6 (meaning between p50 and p60 in this case), January 2 had no spending so it's bucket 1, and Jan 3 had $210 which exceeds p99 ($300)? – Actually $210 is below $300, so it might be bucket 10 if it's >p90 and ≤p99. (This is just illustrative; actual bucket depends on those quantiles.)

We could provide both modes in one response or separate endpoints. Another design is to include both historical and rolling buckets for each day in the data:

```
{
  "userId": 42,
  "days": [
    { "date": "2025-01-01", "total": 34.10, "bucket_historical": 6,
"bucket_rolling": 7 },
    ...
  ],
  "historical_quantiles": { ... },
  "rolling_quantiles": { ... }
}
```

However, this doubles the payload. It might be cleaner to fetch one mode at a time (e.g. an API parameter to choose mode). For our specification, we'll assume separate mode retrieval for simplicity.

Note on database schema: In a SQL database, we might not store JSON but rather a table of daily buckets. The table `heatmap_buckets` could have columns: `user_id`, `date`, `bucket_historical`, `bucket_rolling`, and maybe the `total_spend` (though total could be derived from transactions if needed). Storing the bucket values redundantly is fine since it's a nightly snapshot. The quantile thresholds might be stored in a separate table or as part of metadata (or simply recomputed on the fly each night).

Nightly SQL Generation from `transactions`

We will create/refresh the heatmap data nightly by aggregating the `transactions` table. Pseudocode in SQL (assuming a PostgreSQL-like syntax with window functions for percentile):

```
-- 1. Aggregate daily spending per user
CREATE TEMP TABLE daily_spend AS
SELECT
  user_id,
  date_trunc('day', txn_date) AS date,
  SUM(amount) AS total_spend
FROM transactions
GROUP BY user_id, date_trunc('day', txn_date);

-- 2. Compute quantiles for each user (historical and 1-year rolling)
CREATE TEMP TABLE user_quantiles AS
SELECT
  user_id,
  /* Historical quantiles using percentile_cont (0.1 = 10%, etc.) */
  percentile_cont(0.10) WITHIN GROUP (ORDER BY total_spend) AS p10_hist,
  percentile_cont(0.20) WITHIN GROUP (ORDER BY total_spend) AS p20_hist,
  ...
  percentile_cont(0.99) WITHIN GROUP (ORDER BY total_spend) AS p99_hist,
  /* Rolling 12 months quantiles: filter to last 365 days in the subquery */
  percentile_cont(0.10) WITHIN GROUP (ORDER BY total_spend)
      FILTER (WHERE date >= CURRENT_DATE - INTERVAL '365 days') AS p10_rolling,
  percentile_cont(0.99) WITHIN GROUP (ORDER BY total_spend)
      FILTER (WHERE date >= CURRENT_DATE - INTERVAL '365 days') AS p99_rolling
FROM daily_spend
GROUP BY user_id;

-- 3. Assign buckets by joining and comparing values
INSERT INTO heatmap_buckets (user_id, date, bucket_historical, bucket_rolling)
SELECT
  d.user_id,
  d.date,
  CASE
    WHEN d.total_spend <= q.p10_hist THEN 1
    WHEN d.total_spend <= q.p20_hist THEN 2
    ...
```

```
    WHEN d.total_spend <= q.p90_hist THEN 9
    WHEN d.total_spend <= q.p99_hist THEN 10
    ELSE 10  -- any spend above p99 also gets bucket 10
  END AS bucket_historical,
  CASE
    WHEN d.total_spend <= q.p10_rolling THEN 1
    WHEN d.total_spend <= q.p20_rolling THEN 2
    ...
    WHEN d.total_spend <= q.p90_rolling THEN 9
    WHEN d.total_spend <= q.p99_rolling THEN 10
    ELSE 10
  END AS bucket_rolling
FROM daily_spend d
JOIN user_quantiles q ON d.user_id = q.user_id;
```

This SQL outline does the following: first, compute daily totals. Then, compute quantiles per user; the example uses `percentile_cont` (which interpolates exact percentile values) and shows how to filter for rolling window using a WHERE clause inside FILTER for the window function. Finally, it assigns the bucket by comparing each day's total to the thresholds. We default to bucket 10 for anything above the 99th percentile. The result is inserted into a `heatmap_buckets` table. (In practice, we might `DELETE` and refill this table each night, or maintain it incrementally.)

Data storage considerations: The table `heatmap_buckets` will have one row per user per date (for as many years as in history). With 5 years of data, that's ~1825 rows per user. This is quite manageable, even for many users, and indexing by user_id allows fast retrieval of one user's data. We could also store the quantile values (p10–p99) in a separate table `user_spend_distribution` if needed for reference or for generating legends.

UX Behavior & Component Architecture

Infinite Scroll vs Full Render of Months

The heatmap should support **infinite vertical scrolling by month** – allowing the user to scroll through at least 5 years of data seamlessly. Two implementation approaches are considered:

- **Full Render of 5+ Years:** Render all months' grids in the DOM at once (e.g. 60 months for 5 years). This is simpler but can impact performance if the DOM grows very large. ~60 months × ~30 days = 1800 day cells, plus structure – this is not huge for modern browsers, but if a user has, say, 10-15 years of data (or if multiple users in a single-page app context), it could become heavy. Initial load time might suffer and updating so many elements for theme toggles or window resizing is less efficient.

- **Virtualized Scrolling (Recommended):** Use a virtualization strategy to only render months that are in or near the viewport. Libraries like `react-window` or `react-virtuoso` can create an "infinite" list of month components, mounting/unmounting as needed. This dramatically improves performance by keeping the DOM lightweight [9]. Virtualization ensures smooth scroll and low

memory usage by only rendering visible items [9]. It's scalable even if we allowed decades of data. Given that each month's height is fairly uniform (each month grid will always be 5 to 6 weeks tall), we can use fixed-size virtualization for simplicity: e.g., assume a constant height per month (accounting for the largest case, 6 weeks plus header). This allows an efficient calculation of scroll positions. If we want to be exact, a dynamic height approach (like react-virtuoso) can handle months with 5 vs 6 weeks gracefully as well.

Recommended Implementation: Use a virtualized list of months. For example, wrap the months in a `<Virtuoso>` component (if using React Virtuoso) or use `react-window`'s FixedSizeList. Load, say, the current month and a few ahead/behind initially, and let the virtualization handle the rest as the user scrolls. This way, we support "infinite" history (limited only by data availability) without overwhelming the client. The virtualization approach will keep the UI responsive and memory footprint low [9]. We also gain the ability to easily scroll to a specific month (e.g., jump to current month on load) by using the list's scroll index.

- **Pagination Alternative:** An alternative is manual pagination/lazy-loading: e.g., load 12 months, and when user scrolls near the top/bottom, dynamically append more months. This is doable, but reinventing what virtualization libraries provide. We'd need to manage sentinel elements and state. Using a proven library is less error-prone for features like smooth scrolling and variable heights.

UI Layout: Calendar Grid Design

Each month is presented as a grid of days aligned Sunday–Saturday. We include the month label and day-of-week headers for clarity:

- **Month Label:** At the top of each month section, show the month name and year (e.g. "January 2024"). Style this with Tailwind utility classes (e.g. `text-lg font-bold mt-4 mb-2`). This label can stick to the top of its section. Consider making the month label position:sticky so that as you scroll, the current month label stays at top until the next pushes it out – this way the user always knows which month they're viewing.

- **Day-of-week Header:** A one-row header with Su, Mo, Tu, We, Th, Fr, Sa (or localized variants) should appear above the grid (probably repeated for each month for simplicity, or a single floating header if we wanted – but per-month is fine). Use a smaller font and a neutral color (with good contrast against the background). These can be abbreviated to one or two letters. Use `<abbr title="Sunday">Su</abbr>` so full names are available to assistive tech [6]. Tailwind example: a flex or grid with 7 equally spaced cells, each `text-xs font-medium text-gray-500 dark:text-gray-400`.

- **Days Grid:** The days of the month are laid out in a 7-column grid. We ensure the first day of the month starts under the correct weekday column. One way: include blank filler cells for days before the first of the month. For example, if the month starts on Wednesday, we put two dummy cells for Sunday and Monday (could be just empty placeholders) then start on column 3 for Wed = 1st. This keeps alignment correct. Each week forms a row. We use CSS Grid with `grid-template-columns: repeat(7, minmax(0, 1fr))` so that each day cell takes equal space. Tailwind: `grid grid-cols-7`.

- **Day Cell Design:** Each day is an interactive cell (a button or div). We prefer `<button>` for semantic clickability, with `onClick` opening the details. We apply a Tailwind class for the background color using our computed OKLCH color – since the colors are dynamic per user data, we likely utilize inline style or CSS variables. For example, we might set a CSS variable `--day-color` on each cell and use `style={{ backgroundColor:` oklch(L% C H) `}}` in React, or define some utility classes if the values are discrete. The cell should have a consistent size (we might use a fixed width/height for each cell for a neat grid, or allow responsive sizing). Typically, calendar heatmap cells are small squares or circles. We can use Tailwind padding or aspect-w ratios to enforce a shape. For instance, `aspect-w-1 aspect-h-1 w-8` could make each cell an 8x8 grid box that is square.

- **Text in Cell:** We can display the day of month number in the cell (small text) to help identify the date. Use a short label (e.g. "1" for 1st). This text must contrast with the background (as discussed, black or white depending on theme). We might size it small (e.g. `text-[0.65rem]` or so) and perhaps bold for readability. If the cell is very small, some designs omit the number entirely and rely on tooltips, but given we want at least minimal textual cue and to meet accessibility without color, including the number (or a dot for presence) is good.

- **Today Highlight:** If the design calls for highlighting the current day, we could outline it or add a subtle marker (e.g. a ring). This is an optional UX touch.

- **Tailwind classes example:** A day cell might have classes like:
`focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-blue-500` (for focus highlight),
plus dynamic background color via style or a custom class,
and `transition duration-200 hover:opacity-90` (to slightly highlight on hover).
It might also have `rounded` corners if we want circular-ish cells (a small borderRadius on each cell can make a pleasing design).

Interactions: Click for Details, Long-Press for Annotations

- **Click (or Enter key) – Open Details:** Clicking a day cell opens a detailed view of that day's transactions or summary. This could be a modal dialog or a side panel. In a Next.js app, this might trigger a route change (e.g. `/transactions/2024-01-08`) or simply call a state that displays the transactions. The implementation should focus on performance – possibly prefetch the day's data. The details view should be accessible (if modal, focus trapping, etc.). After closing, focus should return to the cell.

- **Long-Press (or Alt+Click) – Annotate/Budget Edit:** A long press on a day could open an annotation interface. On mobile, we detect a touch hold (e.g. using `onPointerDown` and timing it ~500ms). On desktop, we can interpret a right-click or maybe Shift+Click as the equivalent. Another approach is to have an "Annotate" mode toggle in the UI or an icon on the cell on hover (like a small pencil icon appears that can be clicked). The specification says long-press to edit budget – this implies perhaps the user can set a budget for that day or add a note (e.g. "Trip to NYC – expected high spend"). The UI for this should be a small form (maybe just a text note or a number input for budget) in a popover or modal.

- **Feedback & States:** When a day is clicked or long-pressed, give immediate visual feedback (e.g. the cell could briefly highlight or remain with an "active" state if it stays selected). Ensure that opening a modal from a keyboard action also works (e.g. pressing Enter on a focused day triggers the same handler as click).

- **Tooltips:** It's often helpful to show a tooltip on hover/focus with the date and spend info. For example, hovering a cell could show "Jan 8, 2024 – $45 spent". This duplicates what screen readers hear in aria-label. We can use a lightweight tooltip library or a custom `<div>` that appears on hover. Make sure the tooltip is also accessible (could use `aria-describedby` on focus to tie it to the cell). The tooltip text should also meet contrast guidelines (since it might appear over the cell or at least near it; typically tooltips have their own small bubble with appropriate contrast).

UX Mockup Guidance

Layout & Spacing: The calendar should have some padding around cells and sections. For instance, between month sections maybe a bit of vertical gap (Tailwind `mb-6` after each month). Within a month, the grid cells should be closely packed for a dense heatmap look, but a small gap (`gap-1` or `gap-[2px]`) can be added between cells to separate them visually.

Scrolling Behavior: When the component mounts, you might scroll to the current month automatically (especially if we allow future months too, we might start at "today"). If infinite scroll goes both past and future, ensure the scroll container can handle both directions (some virtualization libraries support prepend). If only past, likely we start at the bottom (current/latest month) and scroll up for history. We should also provide a "jump to today" or "jump to start" button for convenience if the user scrolls far.

Responsive Design: On smaller screens, the grid will shrink. We should ensure the text (day numbers) remain readable (may use slightly larger text on mobile if needed). If the screen is very narrow, maybe we reduce padding to make cells as large as possible. The layout being 7 columns might overflow a very narrow container, but since each cell is likely a fixed min-size, the container can scroll horizontally or we ensure it fits. Ideally, we design it to fit in mobile width by making cells quite tiny but still tappable (maybe minimum 24px touch target). Possibly, on mobile we omit day numbers to make tapping area larger with just color.

Theme Switching: The component should respond to theme changes (light/dark) instantly. Using CSS variables for colors can allow theme switching via a parent class (Tailwind's dark mode toggling). For example, we could define `--heatmap-light-L` and `--heatmap-dark-L` as the base L values, and in CSS define color as `oklch(var(--heatmap-L) var(--heatmap-C) var(--heatmap-H))`. Then toggling the theme could swap those variables. This might be an implementation detail, but ensures the component updates without remount.

Example Tailwind/JSX Snippet:

```
<div className="month-section" aria-label="January 2024">
  <h3 className="text-md font-semibold my-2">January 2024</h3>
  <div className="grid grid-cols-7 text-xs text-gray-500 dark:text-gray-400
mb-1">
    {["Su","Mo","Tu","We","Th","Fr","Sa"].map(day => (
```

```
      <div key={day}>
        <abbr title={fullDayName(day)}>{day}</abbr>
      </div>
    ))}
  </div>
  <div className="grid grid-cols-7">
    {/* leading blanks for alignment, e.g., if Wed = first of month */}
    <div></div><div></div>
    {/* then days */}
    {days.map(d => (
      <button
        key={d.date}
        className="w-8 h-8 rounded focus:outline-none focus:ring-2 focus:ring-
offset-1 focus:ring-blue-600"
        style={{ backgroundColor: d.color }}
        aria-label={`${formatDate(d.date)} – $${d.total} spent, $
{bucketLabel(d.bucket)}`}
        onClick={() => openDetails(d.date)}
        onContextMenu={(e) => { e.preventDefault(); openAnnotation(d.date); }}
      >
        <span className="text-[0.7rem] font-medium
                         text-gray-900 dark:text-gray-100">
          {d.day} {/* day number */}
        </span>
      </button>
    ))}
  </div>
</div>
```

In the above snippet, `d.color` is a precomputed OKLCH string for that day's bucket and month. We used inline style for brevity; in a real app we could also assign a class like `.bg-[oklch(...)]` via a Tailwind plugin or use CSS variables for the color as mentioned.

Keyboard/Focus UX: As the user navigates with arrow keys, when they move into a new month that was previously off-screen (virtualized), the virtualization should load it in time. We ensure the virtualization overscan (render a few items beyond the visible range) so that the adjacent month is present to receive focus. This prevents focus from going into a non-rendered area. If using aria-activedescendant approach (an alternative technique where the grid container has focus and only indicates an active cell), ensure off-screen months can still be referenced (this is more complex; sticking with moving actual focus is easier).

Visual Legend: It might help to display a legend explaining the color intensities. For example, a small bar or set of swatches for buckets 1 to 10 with labels like "≤$5" (p10) up to "≥$300" (p99) could be shown. This gives context to users about what the colors mean. If included, that legend should also meet contrast rules and be accessible (e.g. each swatch labeled).

In conclusion, this financial calendar heatmap component integrates dynamic color scaling tailored to the user's spending distribution, while maintaining accessibility through careful color choices (using OKLCH for

consistent contrast) and proper ARIA/keyboard support. The Next.js + React + Tailwind stack is well-suited for this: we leverage Tailwind for quick styling (grid layout, dark mode support, focus states) and possibly React virtualization for performance. By following this spec, the resulting component will handle large data smoothly and be usable by all users (meeting contrast standards [3] and screen reader navigation patterns).

Sources: The OKLCH color space ensures predictable contrast when adjusting color attributes [1], which we use for hue shifts. APCA guidelines inform our contrast targets (Lc 60 for readable text) [3]. Virtualized scrolling is recommended for performance with large lists [9]. Our keyboard interaction model draws from WAI-ARIA practices for date grids [8]. The combination of these approaches will yield a robust and user-friendly financial heatmap.

---

[1] Using OKLCH colors in Tailwind CSS • Journal • Studio 1902
https://1902.studio/en/journal/using-oklch-colors-in-tailwind-css

[2] Better dynamic themes in Tailwind with OKLCH color magic—Martian Chronicles, Evil Martians' team blog
https://evilmartians.com/chronicles/better-dynamic-themes-in-tailwind-with-oklch-color-magic

[3] [4] [5] How the Washington Post design system made me learn about Perceptual Contrast (APCA) - Steve Frenzel
https://www.stevefrenzel.dev/posts/learning-about-perceptual-contrast/

[6] [7] [8] Date Picker Dialog Example | APG | WAI | W3C
https://www.w3.org/WAI/ARIA/apg/patterns/dialog-modal/examples/datepicker-dialog/

[9] Infinite Scrolling Made Easy: react-window vs react-virtuso | by Stuthi Neal | Medium
https://medium.com/@stuthineal/infinite-scrolling-made-easy-react-window-vs-react-virtuso-1fd786058a73