ChatGPT

Dynamic Color Theming: From Wallpaper to App Palette

Material You's Wallpaper-Based Palette Strategy (Android 12+)

Google's **Material You** (introduced in Android 12) brought a dynamic theming system (code-named "Monet") that extracts colors from the user's wallpaper and generates a full palette for the UI. Unlike traditional design where developers hand-pick a fixed set of colors, Material You uses algorithms to *automatically* create **beautiful, accessible color schemes** based on the image input [1]. The goal is a cohesive, personalized experience – when you change your wallpaper on a supported Android device, **the entire system UI and supported apps recolor to match**, yet text and icons remain legible [2]. This works by deriving a palette of hues from the wallpaper and applying them consistently to UI elements (from backgrounds and buttons to charts and notifications).

At a high level, Google's strategy is: **use the wallpaper as a single source of truth for theming**. The system picks a **"seed" color** (or a small set of key colors) from the image, then algorithmically generates a family of tonal colors (light-to-dark variants) from that seed. These colors are applied across different UI roles (primary, secondary, background, etc.) so that the interface feels unified and tailored to the wallpaper [3] [4]. Crucially, the algorithms ensure sufficient contrast and accessibility. For example, if your wallpaper's dominant color is a light pastel, the system might choose a deeper tone of that color for text and icons so they remain visible, meeting contrast standards [2] [5]. The result is a personalized theme that "just works" across the entire OS and any app that opts in.

Color Extraction: From Image to Key Color Swatches

1. Image Sampling: The process starts with the wallpaper image. Android downsamples the wallpaper (if it's large) to a smaller size (max ~112×112 pixels) before analysis [6]. This optimization makes color extraction faster while still capturing the overall color composition of the image.

2. Quantization (Clustering Colors): The system then runs a **color quantization algorithm** to identify the main colors in the wallpaper. Essentially, this is a clustering process: the algorithm looks at all the pixels and groups them into a limited set of color clusters (palette swatches) based on similarity [7]. Android's Monet engine uses improved *K-Means clustering* algorithms (such as **Variational K-Means** or **Celebi's K-Means** approach) to do this efficiently [8]. If the device has low RAM, it may limit the colors (e.g. 5 clusters); otherwise it can use a finer palette (up to 256 clusters) for accuracy [9] [10].

3. Dominant Color Detection: Once quantization finishes, we have a palette of a few representative swatches from the image. Each swatch knows its color value and how many pixels in the image it represents (its "population") [11]. The swatches are then sorted by population to find the **dominant colors** in the wallpaper [12] [13]. The **largest swatch (most pixels)** usually corresponds to the wallpaper's dominant hue [11]. This dominant color (or colors) is a prime candidate for the theme's core color. (In practice, Monet also

runs a scoring function that can favor certain hues or tones – for example, avoiding overly dark or low-saturation colors as the primary seed – but generally population is a key factor [11] [14].)

4. Choosing the "Seed" Color(s): From the top-ranked swatches, the system picks a **seed color** to generate the theme. Often it's the single most dominant color, unless that color is unsuitable (e.g. too close to pure white/black or low contrast) in which case a secondary color might be chosen. In some cases, multiple seed colors might be used – Material You defines up to **five key hues** (more on this below) – but Android's official logic often simplifies to one source color that everything builds from [3]. Essentially, the extraction phase outputs one or a few **key colors** that will drive the rest of the palette.

> **Note:** This entire extraction is done automatically by the system when the wallpaper changes. Android provides a `WallpaperColors` API that encapsulates the resulting swatches and their stats [14]. The extraction process is designed to be fast and efficient on-device (hence the downscaling and optimized clustering).

Palette Generation: Tonal Palettes and Material You Scheme

After identifying the seed color(s), the next step is to **generate a full palette** that can be applied to the UI. Google's approach is both high-level (defining the roles colors will play) and low-level (computing specific color shades):

- **Five Key Color Roles:** Material You's scheme revolves around *five* core color families: **Primary**, **Secondary**, **Tertiary** (these are the three "accent" colors), plus **Neutral** and **Neutral Variant** [4] [15]. These five are usually derived from the wallpaper's colors – often the primary accent comes from the most dominant hue, while the others either come from secondary image hues or are algorithmically shifted variants (for example, tertiary might be an analogous or complementary hue to add contrast). Monet essentially maps the top-ranked extracted colors to these roles [4]. If the wallpaper is monochromatic, the system will still produce five palettes by tweaking the base color's properties (e.g. reducing saturation for neutrals, or shifting hue slightly for tertiary).

- **Tonal Palettes (Shades):** For **each** of the 5 key colors, Material You generates a **tonal palette** – a set of 13 color shades ranging from very dark to very light [16] [3]. These are typically defined in terms of *tone* (lightness) in the HCT color space (Hue-Chroma-Tone, a perceptually uniform space used by Material You). The tonal palette ensures there are appropriate tints for different UI uses: e.g. a dark variant for text on a light background, a light variant for backgrounds, etc., all maintaining the same hue. In total, the system produces up to **65 color values (5 palettes × 13 tones)** to cover the full range of UI needs [3]. This includes not just straightforward tints, but also "container" colors and outline colors for various elevations in the interface.

- **Ensuring Contrast and Accessibility:** A critical part of the palette generation is making sure the colors work well together and meet accessibility standards. The algorithms will adjust the brightness (tone) and chroma of colors as needed. **For example, in light theme, the primary color is often darkened/desaturated significantly so that white text can sit on it with sufficient contrast** [5]. Similarly, very bright wallpapers won't lead to illegibly bright UI elements – the system's color science ensures the derived palette still has distinguishable, contrasty foreground/background pairs [2]. Material You uses the HCT color model and specific contrast ratios (e.g. aiming for 4.5:1 for text) to

modify the raw extracted colors into **usable UI colors** [5] [17]. The end result is a set of colors that both reflect the wallpaper's spirit *and* maintain readability.

- **Neutral and Accent Separation:** Typically, the most dominant wallpaper color becomes the **Primary** accent color for the theme. Secondary might be a slightly muted or related hue (often a close neighbor of primary) for use in smaller components, and Tertiary is a more contrasting hue (used sparingly for highlights) [18]. The **Neutral** and **Neutral Variant** colors are usually grayish or low-saturation tones derived from the wallpaper's overall color mix – these are used for backgrounds, surfaces, and text, providing a subtle backdrop that doesn't overwhelm. (Sometimes the neutral tones are essentially the wallpaper's color but greatly desaturated) [19]. By having neutrals distinct from accents, the system ensures that backgrounds and high-volume UI areas remain relatively neutral, while accents (Primary/Secondary/Tertiary) provide pops of color for emphasis.

Tonal Palette and Roles Summary: The outcome of the algorithm is a structured color scheme. In Material Design 3 terms, from a single wallpaper **source color**, you get five color families each with a range of tones. These are assigned to roles in the app/theme. For example:

- **Primary** – the main brand or accent color (from wallpaper's dominant hue). Used for key interactive elements (buttons, links, highlights) that you want to draw attention to [18]. (In dynamic schemes, Primary in light mode is often a dark tone of the wallpaper color, and in dark mode it's a lighter tone, to ensure contrast with the surrounding background.)

- **Secondary** – a supporting accent color. Often a variation of the primary hue (or a second dominant hue) but less saturated; used for UI elements of secondary emphasis like filter chips, toggles, etc. It should not compete with Primary for attention [18].

- **Tertiary** – an additional accent, usually with a distinctly different hue from primary/secondary. It's brighter or more vibrant and is used sparingly for **contrasting accents** or critical highlights (for example, an "alert" element or an infographic element might use tertiary) [20].

- **Neutral** – a base neutral color for surfaces and backgrounds. This is typically a soft gray or beige tone influenced by the wallpaper (often the wallpaper's hue but extremely desaturated) [19]. It's used for things like app background, cards, sheets – large areas that shouldn't distract.

- **Neutral Variant** – another neutral (less colorful) tone used for medium-emphasis elements like text, placeholders, or outlines [19]. It provides a second neutral shade (perhaps slightly darker or with a touch of the accent's hue) to use for things like secondary text or strokes, so everything isn't the exact same gray.

Each of these roles will have a specific tone assigned depending on light/dark mode and usage context (e.g. "Primary color" vs "Primary **Container** color" which is a lighter variant used for filled components). Material You defines these systematically. The key point is that **the palette covers the full UI spectrum** – from backgrounds and surfaces (neutrals) to vibrant accents – all derived from one wallpaper. Developers and the system use these as named colors (color tokens) rather than hard-coded hex values [17], which is why the scheme can adapt to any image input and still maintain design consistency.

Applying the Palette Across UI and Data Viz

Once the dynamic palette is generated, Android applies it **everywhere** it can for a unified look:

- **System UI:** The status bar, quick settings, notifications, volume sliders, etc., all get tinted with the dynamic colors. Android uses a **Theme Overlay** mechanism to apply the new colors to system UI components whenever the wallpaper (and thus palette) changes [21] [22]. For example, the Quick Settings tiles might use the Primary color for their active state, or the notifications shade might use a neutral variant as its background.

- **Apps (Material Components):** Google made it easy for apps to adopt dynamic color. The generated color scheme is exposed via APIs so that apps can retrieve the Material You colors [23]. In Android 12+, if an app uses Material Design 3 (Material You) components or theme, it can simply enable dynamic color and the framework will apply the user's wallpaper-derived colors to the app's UI controls automatically. Under the hood, the 65 color attributes (the tonal palettes) are available for apps – e.g. `colorPrimary`, `colorOnPrimary`, `colorSurface`, etc., are filled with the dynamic values [24]. This means if your app's UI is built using those semantic color roles, it will *automatically* switch to match the wallpaper theme, creating that "flawless" cross-app cohesion.

- **Data Visualization and Other Elements:** Because the dynamic palette provides multiple accent colors and neutral tones, developers can also apply them to custom visuals – for instance, a chart or graph in an app could use the Primary, Secondary, and Tertiary colors for its series to match the overall theme. The neutrals can be used for chart gridlines or backgrounds. Since the Material You palette is designed to have good contrast among its tones, using them in data visualizations tends to work well (e.g. a tertiary accent line on a neutral background will be clearly visible). Essentially, the palette is broad enough to color not just standard UI widgets but any custom UI element. Developers are encouraged to use the dynamic **color roles** (primary, secondary, etc.) for any content in their app that should feel integrated with the system's look [24]. This is how, for example, even third-party apps (like Google's own apps: Gmail, Photos, etc.) achieve that Material You styling – by applying the dynamic palette to their toolbars, icons, and even illustrations or charts inside the app.

To summarize the application strategy: **the palette is injected into every layer of the UI**. Android's system picks up the colors and **saves them in a shared resource index accessible via API to all apps** [23]. Apps that want to participate simply fetch those colors (or use Material library support) and apply them to their UI elements. The result is a consistently themed experience – you'll notice buttons, backgrounds, and even data visuals in different apps all harmonize with whatever wallpaper you've chosen, without each app developer manually designing a color scheme for every possible wallpaper. The combination of a well-defined palette (with roles and tonal variants) and platform support for propagating those colors is what makes it "just work" across the UI.

Implementing a Similar Theming in iOS (Swift)

You mentioned you want to do this in an **iOS app (Booksmaxxing)** – generating a theme from a book cover image and applying it to your app's lessons and progress views. While iOS doesn't have a built-in dynamic wallpaper theming engine like Android's Monet, you can achieve a similar effect with a bit of work. Here's how you might approach it:

1. Extract Key Colors from the Image: Use an algorithm/library to get the dominant colors from the book cover. There are open-source options available. For example, **ColorThief** (originally a JS library by Lokesh Dhakar) has a Swift port called **ColorThiefSwift** that can grab the dominant color or a representative palette from an image [25]. With such a library, you can feed in a `UIImage` of the book cover and retrieve either the single most prominent color or a set of top colors. Alternatively, you could use Apple's Vision or CoreImage to cluster colors, but using a ready library is simpler.

For a more faithful reproduction of Material You's approach, Google actually open-sourced their **Material Color Utilities** – which include the color extraction and scheme generation algorithms – with support for multiple languages including Swift [26]. This means you can use Google's code to do everything Monet does, right in your iOS app. The Material Color Utilities library provides components for **quantizing an image's pixels into color clusters, scoring/ranking those clusters, and generating a Material Design 3 color scheme** from a seed color [27]. You could, for instance, use its image quantization to get the top colors of the cover, then use its Scheme generator (which uses the same HCT tonal palette logic) to create a full set of tones (primary, secondary, etc.). This library ensures the resulting colors meet the contrast requirements and adhere to Material's scheme logic out of the box.

2. Generate a Theme Palette: Once you have one or more dominant colors from the cover, decide on your palette. If using the Material utilities, you can input a single seed color and get back a `ColorScheme` object (with all the roles: primary, onPrimary, primaryContainer, secondary, etc. for light/dark modes). If you go with a simpler approach (e.g. just ColorThief), you might retrieve say 5 top colors and then manually assign them to roles (e.g. pick the most dominant as primary, another as secondary, maybe use one for background if it's low saturation, etc.). In either case, you'll likely want to **create tonal variations** of those colors. On iOS/Swift, you can do this by converting to HSB or LAB color space and adjusting brightness. (Material's library would handle tonal variation for you via its HCT model). The key is to produce a set of colors to cover: primary accents, secondary accents, tertiary (if needed), plus background and surface colors, and maybe a distinct text color if your background is dark. Ensure to check contrast – for instance, if you generate a background color from the image, you might use a very light tone of a color so that dark text is readable on it, or vice versa.

3. Apply the Colors to Your UI: With the palette in hand, update your app's UI elements to use these colors. In SwiftUI, you might define a custom `ColorScheme` or use `Theme` singletons. For example, you can create Color constants for primary, secondary, background, etc., and use them throughout your views. If your app is structured to support theming, you could inject these as environment values or via a Theme manager. In UIKit, you might set the `tintColor` of your views, navigation bars, buttons, etc., to the new primary color. For things like progress views or charts, apply the accent colors from the palette (e.g. use primary color for the main progress bar, maybe secondary color for other progress or decorative elements). Because you have a **neutral variant color**, that could be great for text or secondary labels; a **neutral background** color can be applied to card backgrounds or collection view cells so the overall view takes on a subtle tint from the book cover. Essentially, wherever you previously used fixed colors, switch to use the dynamic ones. Many iOS developers create a `Theme` struct or use Asset Catalog named colors that they programmatically set at runtime.

One thing to watch on iOS is ensuring the colors update when the book changes. You might need to trigger a UI refresh (in SwiftUI this could be as simple as binding the colors to a published state; in UIKit you might need to apply them and call `setNeedsLayout`). Unlike Android, which seamlessly applies themes via

system, on iOS you control when to apply (for example, when the user selects a book, you'd run the extraction and then update the UI).

4. Maintain Harmony and Accessibility: Try to mirror Material You's tactics for a polished result. For instance, if the book cover's dominant color is very bright, consider using a slightly darker version of it for large areas so that white text will be readable on it (just as Material You does by darkening primary for light mode). Likewise, ensure your text color is either very close to white or black (depending on background) for maximum contrast – you can decide this by checking the luminance of your background color. The Material color utilities library can actually compute "contrast aware" colors for you (it has a contrast module) [28], or you can use WCAG contrast formulas. The end goal is that no matter what cover image is thrown in (be it a deep black cover or a vibrant multicolor cover), the generated theme still looks good and all text/data remains easy to read. A bit of testing with diverse covers will help fine-tune your implementation.

5. (Optional) Leverage iOS 15+ Features: Apple doesn't have dynamic theme by image, but it does have dynamic **dark/light mode** colors. If your app supports both light and dark mode, you should generate a separate palette for each (Material You does this too). For example, the same seed color can produce a lighter variant for dark mode backgrounds. You can supply colors using the asset catalog that vary by *appearance* (dark vs light). In SwiftUI, you might simply choose appropriate tones depending on `ColorScheme` environment.

In summary, implementing this on iOS will involve picking a library for color extraction (Google's Material Color Utilities in Swift is a great choice for a ready-made solution, as it gives you the full Material You pipeline [27]). You'll extract the dominant color from the book cover, generate a palette (primary, secondary, tertiary, neutral, etc. with various tones), then programmatically apply those to your app's UI components and charts. The result should be a dynamic, book-specific theme – very much analogous to Android's wallpaper-based theming, but tailored to your context (book covers). By following the strategy and tactics Google used (clustering image colors, ranking them, creating tonal palettes, and assigning them to semantic roles in the UI), you can achieve a visually appealing and **cohesive look** for each book that feels natural and ensures usability [2].

Sources:

- Google Material Color Utilities (open-source algorithms in Swift for dynamic color) [1] [27]
- Android 12 Monet engine – color extraction and palette generation internals [7] [11] [4]
- Material Design 3 dynamic color guidance (5 key colors and tonal palettes) [15] [5]
- gHacks Tech News – Overview of Monet color theming system [29] (5-color palette with 12 shades each, accessible via API)
- Mike Booth Design – "Monet" color personalization (legibility and contrast emphasis) [2]
- FiveDotTwelve Blog – Dynamic Color UX (accent/neutral roles and usage tips) [15] [30]
- ColorThiefSwift library (for color extraction on iOS) [25]

---

[1] [26] [27] [28] GitHub - material-foundation/material-color-utilities: Color libraries for Material You
https://github.com/material-foundation/material-color-utilities

[2] Recreating Android's 'Monet' colour schemes in your web app - Mike Booth Design
https://www.mikeboothdesign.co.uk/recreating-androids-monet-colour-schemes-in-your-web-app/

3 24 Material You design | Android Open Source Project
https://source.android.com/docs/core/display/material

4 6 7 8 9 10 11 12 13 14 16 21 22 Chasing Monet inside the Android framework · Sid Patil
https://siddroid.com/post/android/chasing-monet-inside-the-android-framework/

5 15 17 18 19 20 30 Dynamic Color on Android from a UI/UX designer point of view – FiveDotTwelve – Full-cycle Flutter app development company
https://fivedottwelve.com/blog/dynamic-color-on-android-from-a-ui-ux-designer-point-of-view/

23 29 Material You's dynamic wallpaper theming system may arrive with Android 12.1 - gHacks Tech News
https://www.ghacks.net/2021/09/23/material-you-dynamic-wallpaper-theming-system/

25 GitHub - yamoridon/ColorThiefSwift: Grabs the dominant color or a representative color palette from an image. A Swift port of Sven Woltmann's Java implementation.
https://github.com/yamoridon/ColorThiefSwift