How to update styles, texts, colors, and images, icons directly on the UI thread (60fps) using RN Reanimated 3

12 min read · Jan 27, 2024


Varun Kukade Follow

Note:- Do read this article written by me as changes in this article will be built on top of that.

Let's start with fundamental concepts:

What are Program and Process — You are visiting this page through Google Chrome and Google Chrome is one of the programs where multiple processes are running inside this program. That means Google Chrome is multi-processed. For example — Downloading files from some webpage is one such process and creating a new tab and visiting a new webpage is another process.

What is Thread — For each of the processes either single or more than 1 thread is used.
Single-threaded means only 1 thread is used for execution of that process which further means the execution of diff tasks inside the process cannot happen in parallel. They only happen sequentially.
Multi-threaded means more than 1 thread is used for the execution of a process which further means more than one task can be executed at a time.

Example —
Google Chrome has various processes running. One of them is the process of downloading files from the internet to the desktop.
You started 1st file download: Chrome allows it and now it's in progress.

Before finishing 1st download, you start 2nd file download: Here chrome allows it and starts the 2nd file download.
Here if you see google chrome downloading process allows execution of more than 1 task simultaneously. That's why we can say this process of downloader as multi-threaded.

React Native — React native has various processes under it. That means it is multi-processed. Now here each process is assigned only 1 thread. That means inside any process 2 things cannot run simultaneously. That means it is single—threaded.

Mainly it has 2 processes. —

1. **UI rendering**: This process has been assigned a thread called UI thread. UI thread handles layout calculations, renders the UI on the screen, and re-renders/recalculates UI elements. The UI thread is also handling virtual dom.

   Virtual dom — React native has a main dom tree which includes various objects representing actual UI elements with every object having various styles, props, etc. Now when we change some styles/props, we are expecting to see the change on the UI screen. React native takes help from Virtual Dom in this case. When we change some styles, react native creates the virtual copy of the main dom called virtual dom with new changes. Now, the virtual dom object is compared with the main dom object. Due to this, React native knows the exact changes that need to be done inside the main dom object. Now React native only re-render the section of UI which is changed instead of re-rendering the complete dom tree
   Virtual dom is just an object and is used for calculations only. It is not used for rendering anything. The main dom tree object is still used for rendering UI.

2. **Executing JS**: This process has been assigned a thread called JS thread. JS thread takes care of all JS calculations/code. React native uses JS core as the default engine for Android and IOS.

JS thread and UI thread work together to display and render something on the UI screen. There are two ways animations can be executed here. The first is completely through the JS thread and the second is completely through the UI thread.

1. **Animation through JS thread —**
   As I explained in this article, if we want to animate the translateX value from 0 to

20, we can store the initial value 0 in useState and update the state.
That means for every updated value 0, 0.01, 0.02, 0.03 up to 20, the component will be re-rendered and that is a performance issue as the component is getting rerendered a lot of times.
As useState lies on the javascript thread, we are doing everything on the javascript thread. In case we need a javascript thread for other calculations, then we will experience slowness in the app as the javascript thread has to handle both javascript calculations and also animation at the same time.
The same problem can happen for text changes. Let's say we have some text to display on the screen and we have a pan gesture slider or something like this. On the pan gesture of the slider in x direction, we have to update the text frequently. In that case, storing the text in useState and updating it frequently will cause a performance drop.
If we update the state, javascript does the calculations and updates the useState. Now to see the updated value on the screen, the JS thread sends the updated value to the UI thread through the bridge. Imagine a bridge between UI and JS thread and communication and data exchange happens through this bridge. Now as we can imagine bridge will introduce overhead and complexity and will take some time to exchange, serialize, deserialize, and transfer the data.
Hence if we update the useState very frequently it will involve a lot of to-and-fro data exchanges between javascript and UI threads. Hence it will cause a lot of delay and animation would be laggy.
To perform the animation on the JS thread we can make use of Animated API from react native. Animated API also provides a way to execute animation on a UI thread using a native driver. But it's limited to some properties only.

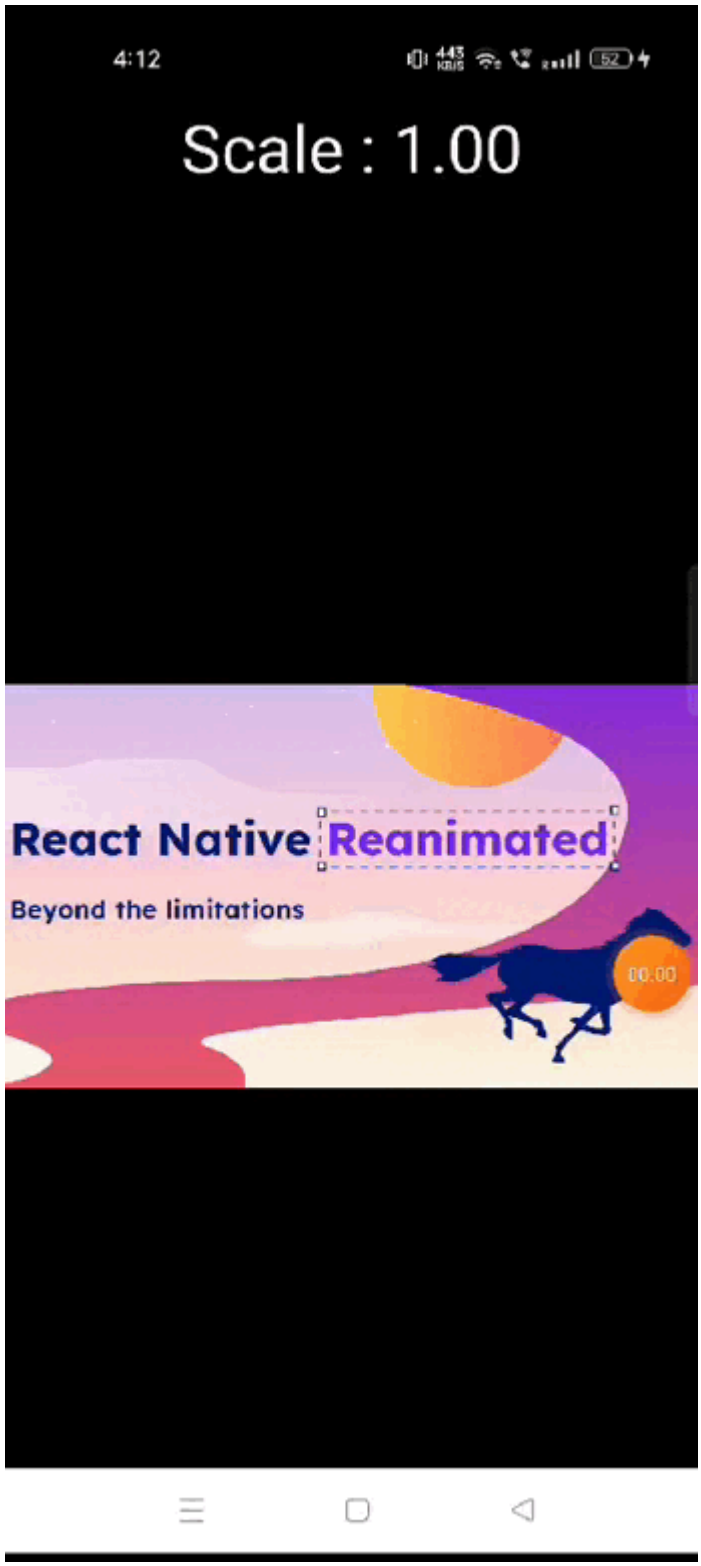
2. **Animation through UI thread —**
   React native reanimated solves this problem. It does all the calculations on the UI thread itself. It spawns the new JS thread on the UI thread and hence does all JS and UI calculations on the UI thread itself. Hence there is no to and fro data exchange and no delay. Through reanimated, in the process of animation, there is not a single re-render on the JS side and animation feels a lot smoother.
   Hence reanimated gives us the capability to create the state on the UI thread using the useSharedValue hook. Hence even if we change the shared value, no re-render occurs.
   Also, functions are created on the UI thread itself. These functions are termed

worklets.
Any updates to the useSharedValue variables should be done from worklets only.

Now let's see one by one how we can update styles, texts, colors, and images directly on the UI thread without a single re-render on the JS thread.

1. **Styles** — this article covers the style changes using reanimated.
2. **Texts** — Let's see how to change the text UI thread. We will add more code on the top of the Animated Image that we covered in this article.

Here's a final demo video of how text changes would look like:



Right now there is no way we can use the React native core Text component and change it on the UI thread. But as we have useAnimatedProps from reanimated, we need some component where we can pass the text as property. I will explain useAnimatedProps in some time.

```
export const AnimatedTextInput = Animated.createAnimatedComponent(TextInput);
```

Here we will create Animated Text input component by passing it to createAnimatedComponent. We would be using AnimatedTextInput to display the text.

```
<Animated.View style={styles.scaleTextContainer}>
   <AnimatedTextInput
      underlineColorAndroid="transparent"
      editable={false}
      style={styles.scaleText}
      animatedProps={scaleAnimatedProps}
   />
</Animated.View>
```

First whichever component we need to animate, we need to wrap it with <Animated.View>.
Then we used AnimatedTextInput and added some styles to it to make it look like same as Text component.

For the animated components, if we need to animate/change styles on the UI thread, we generally pass styles in style prop. Now if we need to animate/change any other props, reanimated gives us the animatedProps prop to pass down the animated props. As you can see we passed scaleAnimatedProps which is an animated props object. Here's how we can create this object.

```
const scaleAnimatedProps = useAnimatedProps(() => {
    return {
      text: scaleTextDerived.value,
      defaultValue: scaleTextDerived.value,
    };
  });
```

We need to use useAnimatedProps hook from reanimated. It lets you create an animated props object that can be animated using shared values.

Here we passed text and defaultValue saying this put this text into textinput. Here useAnimatedProps would recreate the scaleAnimatedProps every time

scaleTextDerived shared value changes. Hence text would every time contain the latest value that scaleTextDerived returned.

Now let's see what is scaleTextDerived.

```
const scaleTextDerived = useDerivedValue(() => {
    //this function will be run whenever scale change
    if (scale) {
      return `Scale : ${scale.value.toFixed(2)}`;
    } else {
      return '';
    }
  });
```

useDerivedValue lets you create new shared values based on existing ones while keeping them reactive. Now here callback function passed to the useDerived value will run every time the shared value used inside the function changes. Hence scaleTextDerived shared value will always be updated every time we have some change in scale shared value.
This callback will automatically be changed into worklet and will run on the UI thread itself.

Hence now whenever we zoom in or zoom out, the scale will be increased in pinchGestureHandler and scaleTextDerived will be recreated based on the new scale value. Now useAnimatedProps will recreate the scaleAnimatedProps object based on the latest scaleTextDerived value and we will see the latest scale value on the UI.

Last step:- Reanimated supports various styles and properties of react native core components for animation default. But reanimated doesn't support the text property of TextInput to be animated by default. Hence we would need to tell reanimated to support it by using the following.

```
Animated.addWhitelistedNativeProps({text: true});
```

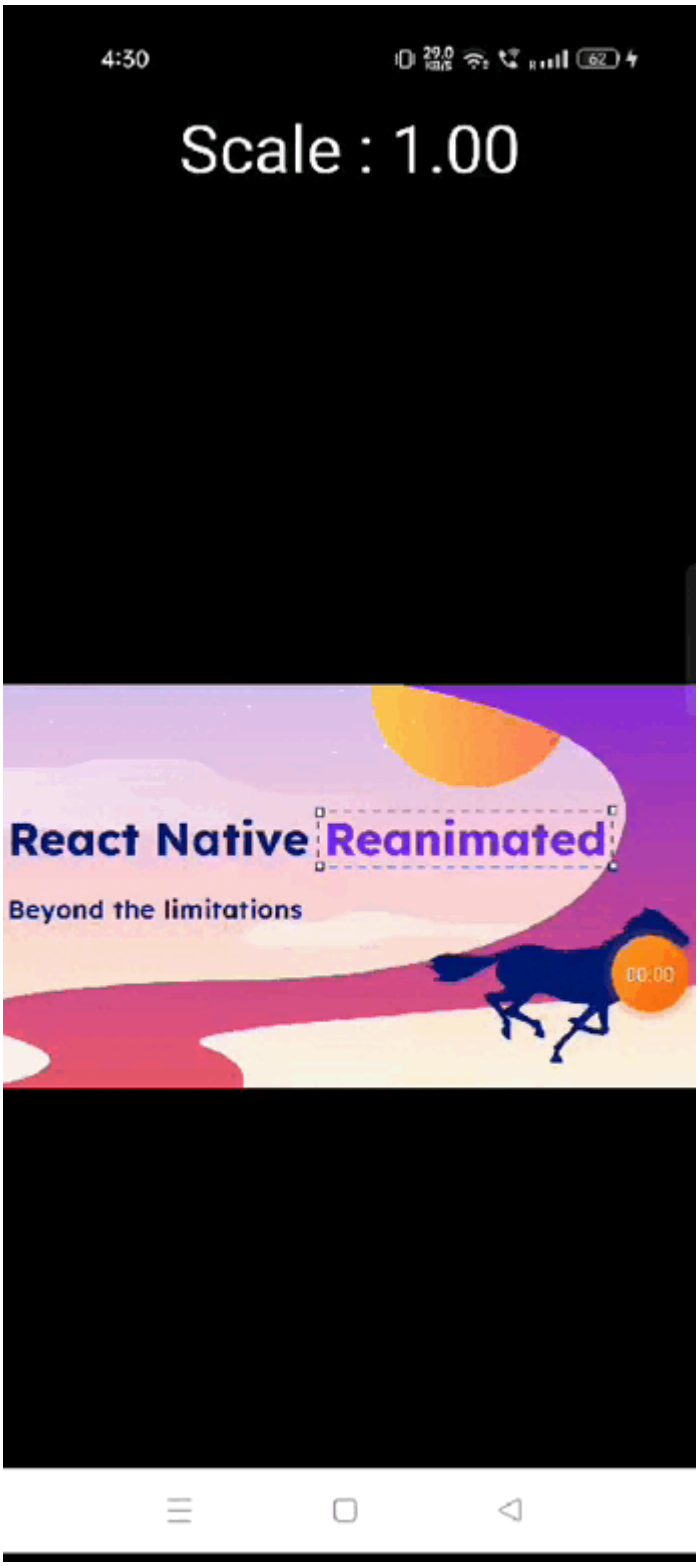
Make sure to add it outside of the functional/class component. By using addWhiteListedNativeProps we add text property to the native props list and hence now reanimated would consider it for animation.
You can find the list of native props supported by reanimated here.

Adding a property to the addWhiteListedNativeProps works only if that specific property is used/present at the native side directly. As React native uses text property at native side with TextInput we were able to animate it. In the following scenarios, this won't work:

1. Let's say we add some random/unknown property by using addWhiteListedNativeProps and it's not present on the native side at all, then the animation won't work.
2. Let's say we add one known property for third-party component. But if that specific known property is only used as a prop to that component and is parsed on the JS side and is not present at the core native side, the animation won't work.

3. **Colors** — Now let's see how we can change colors on the UI thread

Here's a final demo video of how color changes would look like:



As you can see when we zoom in, the background color of the image changes.

Let's wrap our component with Animated.View

```
<Animated.View style={[styles.container, animatedBgColorStyle]}>
      <Animated.View style={styles.scaleTextContainer}>
        <AnimatedTextInput
          underlineColorAndroid="transparent"
          editable={false}
```

```
            style={[styles.scaleText, animatedTextColorStyle]}
            animatedProps={scaleAnimatedProps}
          />
        </Animated.View>
        <View style={styles.imageContainer}>
          <GestureDetector gesture={composed}>
            <AnimatedFastImageComponent
              {...props}
              style={[animatedStyle, style]}
              resizeMode={resizeMode}
            />
          </GestureDetector>
        </View>
      </Animated.View>
```

Here is how animatedBgColorStyle looks like —

```
const animatedBgColorStyle = useAnimatedStyle(() => {
    return {
      backgroundColor: interpolateColor(
        scale.value,
        [MIN_SCALE, MAX_SCALE],
        ['black', 'grey', 'white'],
      ),
    };
  });
```

interpolateColor is a function provided by reanimated which interpolates the colors based on the provided arguments. Here it accepts the shared value, input array, and output array.

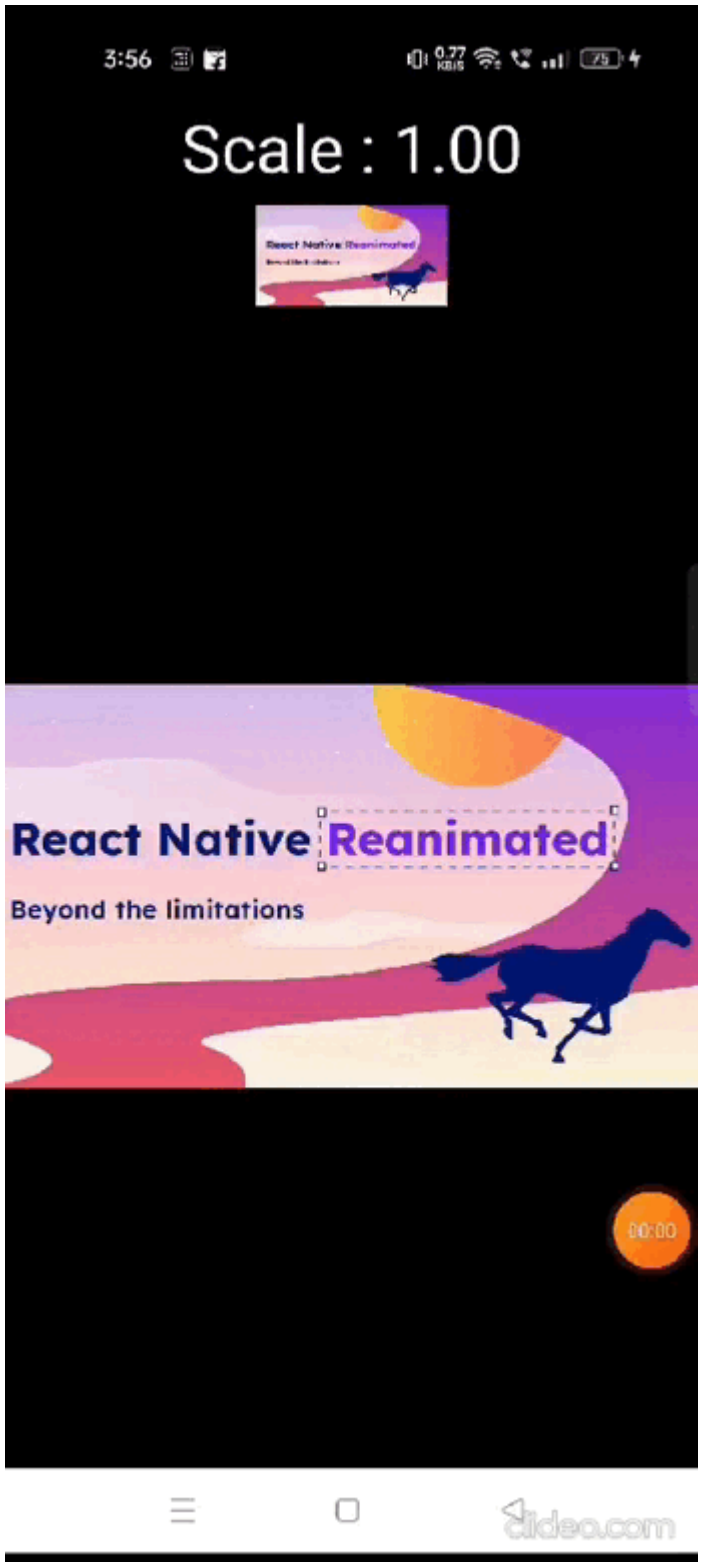
1. We need to change colors as we scale up or down. Hence I passed scale.value.
2. Now we need to change colors when the scale value changes from MIN_SCALE (1) up to MAX_SCALE (2).
3. Now we need to have black color at MIN_SCALE and as we scale up from min scale to MAX_SCALE, we need to change color to white. We need to add at least 2 colors as we have 2 inputs in the input array.
   Note that this only accepts the colors provided by React native.

We can change text color in the same way: -

```
const animatedTextColorStyle = useAnimatedStyle(() => {
    return {
      color: interpolateColor(
        scale.value,
        [MIN_SCALE, MAX_SCALE],
        ['white', 'black'],
      ),
    };
  });
```

4. **Images** — Now let's see how we can change images directly on the UI thread.

Here's how the final demo of image changes would look like



This was challenging for me. I tried to update the images directly on the UI thread by changing the source prop using useAnimatedProps. But that didn't work.

The core Image component was not working even after involving the js thread as I noticed some flickering issues with it when images get updated.

I tried the option such as expo-image. expo-image worked without any flickering, but it involved one trip to the JS thread.

Also, react-native-fast-image worked on IOS by preloading images using the preload method. But it also involved one trip to the JS thread. As we know preload method accepts the URI and if you have local images, you will need to convert the local image path into a remote URI. For IOS I converted the local image path to URI using

Hence I decided to use one hack which worked as expected.

As a first step, I loaded some images and put them in a constants file as follows:

```
export const images = {
  1: require('../assets/images/reanimated.png'),
  2: require('../assets/images/reanimated2.png'),
  3: require('../assets/images/reanimated3.jpeg'),
  4: require('../assets/images/reanimated4.jpeg'),
  5: require('../assets/images/reanimated5.png'),
  6: require('../assets/images/reanimated6.jpeg'),
  7: require('../assets/images/reanimated7.jpeg'),
};
```

Then I created one shared value for storing the current displayed image ID. I updated the image id in the gesture handler while performing the pinch gesture.

```
const currentImageId = useSharedValue(1);

const pinchGesture = useMemo(
    () =>
      Gesture.Pinch()
        .onUpdate(e => {
          scale.value = savedScale.value * e.scale;
          if (scale.value < 1.5 || scale.value > 2.7) {
            currentImageId.value = 1;
          }
          if (scale.value > 1.5 && scale.value < 1.7) {
            currentImageId.value = 2;
          } else if (scale.value > 1.7 && scale.value < 1.9) {
            currentImageId.value = 3;
          }
          if (scale.value > 1.9 && scale.value < 2.1) {
            currentImageId.value = 4;
```

```
          }
          if (scale.value > 2.1 && scale.value < 2.3) {
            currentImageId.value = 5;
          }
          if (scale.value > 2.3 && scale.value < 2.5) {
            currentImageId.value = 6;
          }
          if (scale.value > 2.5 && scale.value < 2.7) {
            currentImageId.value = 7;
          }
        })
        .onEnd(() => {
          savedScale.value = scale.value;
          if (savedScale.value < MIN_SCALE) {
            //do not allow zooming out below original/min scale
            scale.value = withSpring(MIN_SCALE, {
              stiffness: 60,
              overshootClamping: true,
            });
            savedScale.value = MIN_SCALE;
          } else if (savedScale.value > MAX_SCALE) {
            //do not allow zooming beyond max scale
            scale.value = withSpring(MAX_SCALE, {
              stiffness: 60,
              overshootClamping: true,
            });
            savedScale.value = MAX_SCALE;
          }
        }),
    [scale, savedScale, currentImageId],
  );
```

Then as the next step, I rendered all images on top of each other by using absolute positioning.

```
{Object.entries(images).map(eachItem => {
        return (
          <Animated.View
            key={eachItem[0]?.toString()}
            style={styles.image2Container}>
            <AnimatedFastImageComponent
              source={eachItem[1]}
              resizeMode={'contain'}
              style={[styles.image2, getOpacityStyle(eachItem[0])]}
            />
          </Animated.View>
```

```
        );
      })}
```

Now even if all images are present on top of each other, I want to only show/render the image whose ID matches with the currentImageId shared value. Hence I dynamically changed the opacity in the UI thread using useAnimatedStyle.

```
const getOpacityStyle = (id: string) => {
   return useAnimatedStyle(() => ({
     //change opacity in UI thread.
     opacity: id.toString() === currentImageId.value.toString() ? 1 : 0,
   }));
 };
```

Hence now if we zoom in /zoom out the original image, we can see that the image with the matching ID for currentImageId will be visible.

For remote images, you can use the same strategy.

```
export const httpImages = {
  1: 'https://docs.swmansion.com/react-native-reanimated/img/og-image.png',
  2: 'https://user-images.githubusercontent.com/16062886/117443145-ff868480-af3
  3: 'https://i.ytimg.com/vi/U_V9pHnTXjA/maxresdefault.jpg',
  4: 'https://www.reactnativedevelopmentcompany.co.uk/wp-content/uploads/2023/0
  5: 'https://miro.medium.com/v2/resize:fit:1400/1*-A7uP61vaSfKMykGO9Z-PQ.png',
  6: 'https://i.ytimg.com/vi/kAn-Zn8v5kc/maxresdefault.jpg',
  7: 'https://i.ytimg.com/vi/jvIuZrLgbMw/maxresdefault.jpg',
};
```

You just need to use the uri property in the source.

```
      {Object.entries(httpImages).map(eachItem => {
        return (
          <Animated.View
            key={eachItem[0]?.toString()}
            style={styles.image2Container}>
```

```
            <AnimatedFastImageComponent
              source={{uri: eachItem[1]}}
              resizeMode={'contain'}
              style={[styles.image2, getOpacityStyle(eachItem[0])]}
            />
          </Animated.View>
        );
      })}
```

I also tried this approach with more than 15 images of almost 1 mb size for each image and it worked as we are not rendering anything. Every image is already present there and no need of unmounting and mount image.

5. **Icons** — We can use the same strategy of placing icons on top of each other using absolute positioning and then dynamically showing the icon similar to images.

That's it for this article. Here's the final repo link — https://github.com/varunkukade/Animated-Image

If you found this tutorial helpful, don't forget to give this post 50 claps 👏 and follow 🚀 if you enjoyed this post and want to see more. Your enthusiasm and support fuel my passion for sharing knowledge in the tech community.

I have already covered wide range of topics on react native. You can find more of such articles on my profile -> https://medium.com/@varunkukade999

Stay tuned for more in-depth tutorials and insights on React Native development.

React Native | React Native Development | React Native Developers | Mobile App Development | Mobile App Developers

Follow

Written by Varun Kukade

390 followers · 27 following

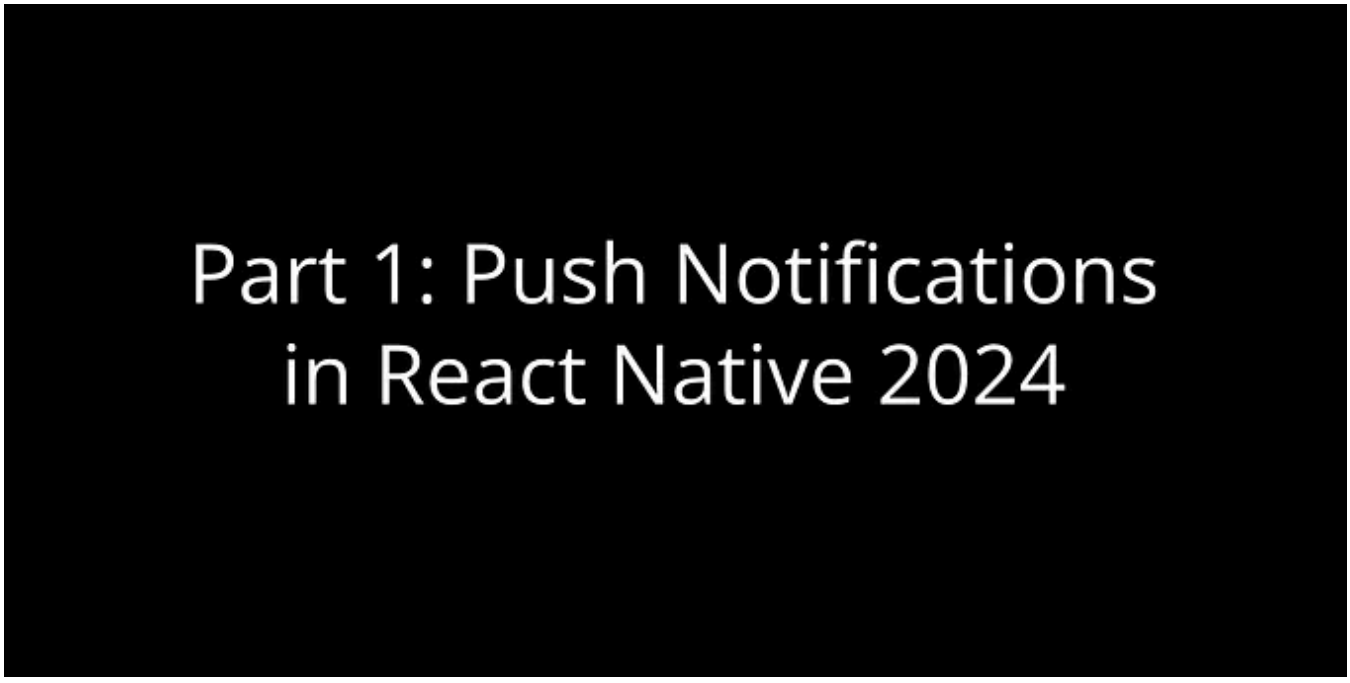
Mobile Engineer 🚀 https://github.com/varunkukade varunkukade999@gmail.com

No responses yet

Write a response

What are your thoughts?

More from Varun Kukade



Varun Kukade

Part 1: Push Notifications in React Native 2024

I recently had a chance to deep dive into Push notifications at my workplace and learned a lot of things. I had to start with what is push...

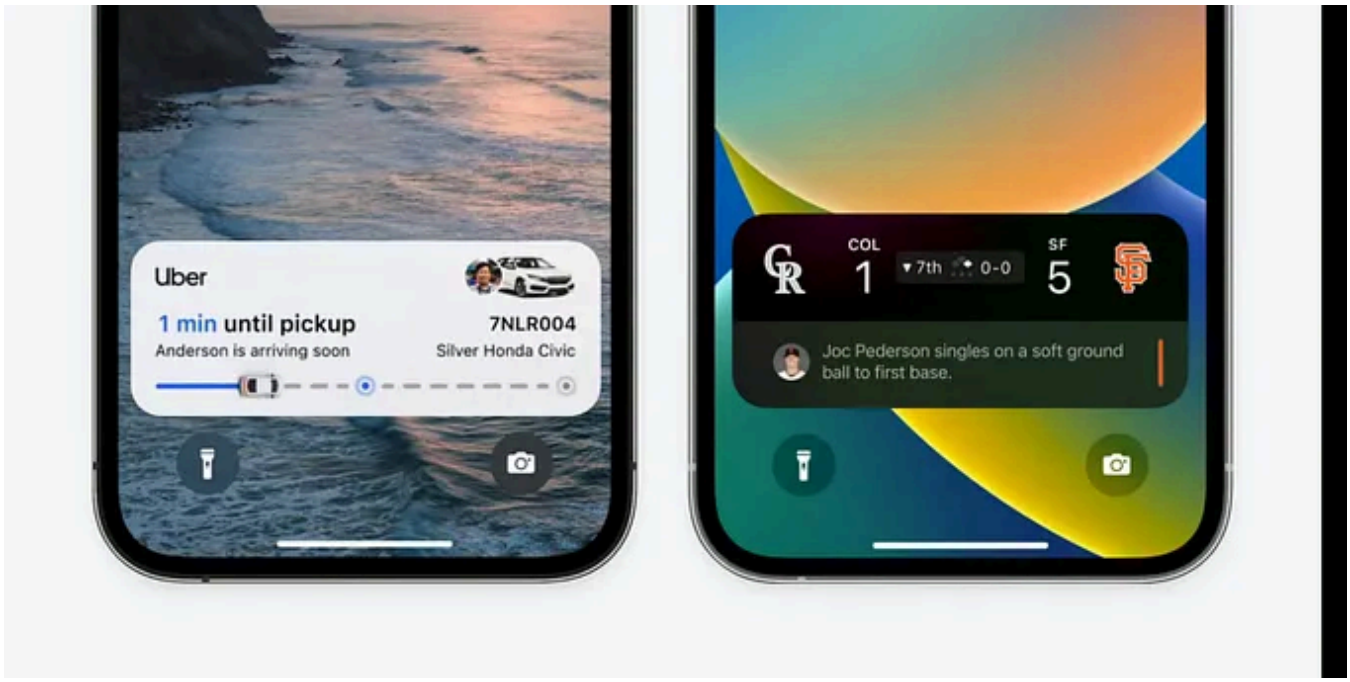
Sep 29, 2024 531 8

Varun Kukade

Part 1: Build your first AI agent + chatbot with Langchain and LangGraph in Python

In this article, we'll learn how to build a simple AI chatbot + agent using LangChain and LangGraph. If you're already familiar with AI...

Jul 20, 2025 27 2



Varun Kukade

Part 1: IOS Live Activities in React Native

At my workplace, Altir, I recently integrated IOS Live Activities into one of our core products, Hyperfuel, using React Native. During the...

Apr 28, 2025 155 1

Varun Kukade

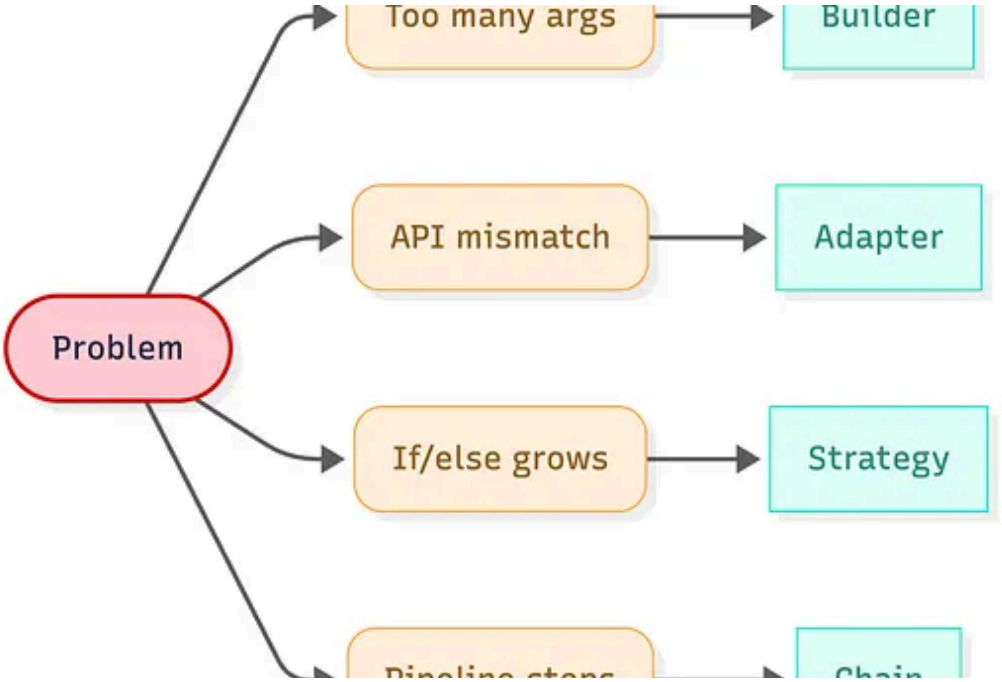
QR code scanner in React Native

In this article, we will see how to create a QR code scanner in react native.

Mar 17, 2024 296 3

See all from Varun Kukade

Recommended from Medium



In Women in Technology by Alina Kovtun ✨

Stop Memorizing Design Patterns: Use This Decision Tree Instead

Choose design patterns based on pain points: apply the right pattern with minimal over-engineering in any OO language.

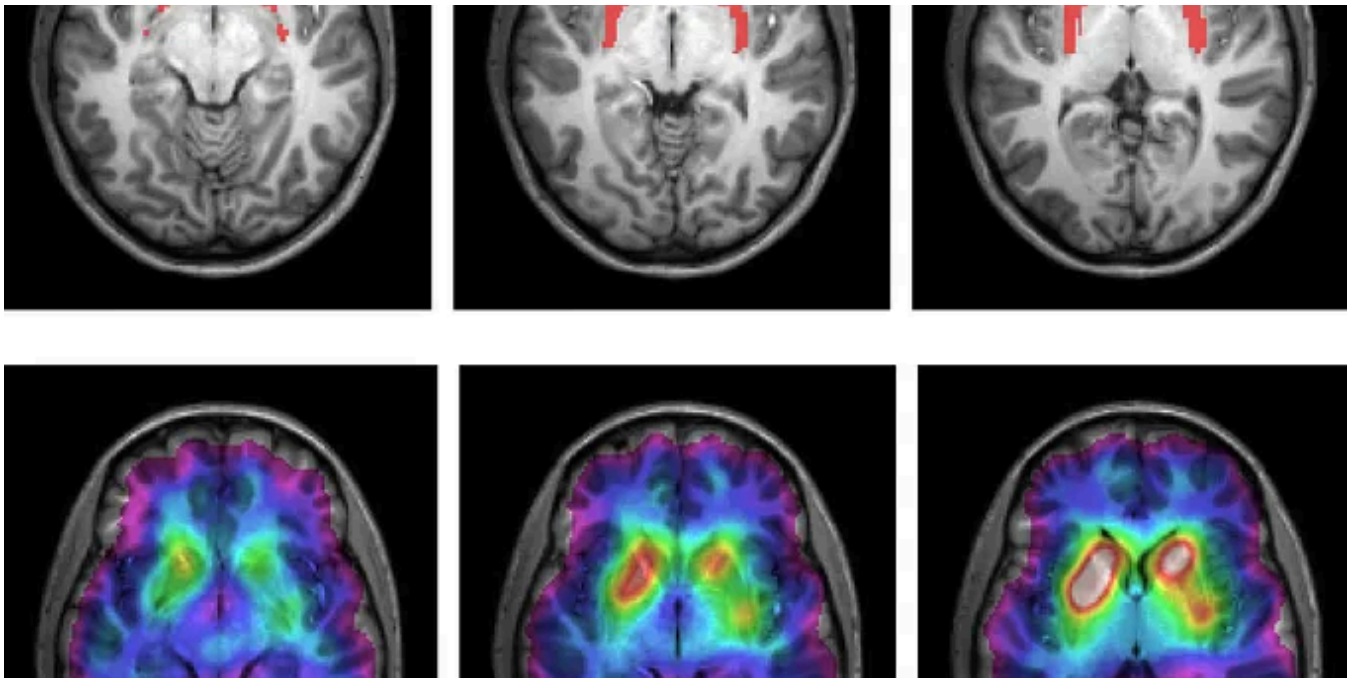
Jan 29 1.7K 12

Shanavas Shaji

Upgrading to Expo 54 and React Native 0.81: A Developer's Survival Story

When Expo 54 dropped, I didn't jump in just for the shiny new features. I had three pressing reasons:

Sep 12, 2025 151 2

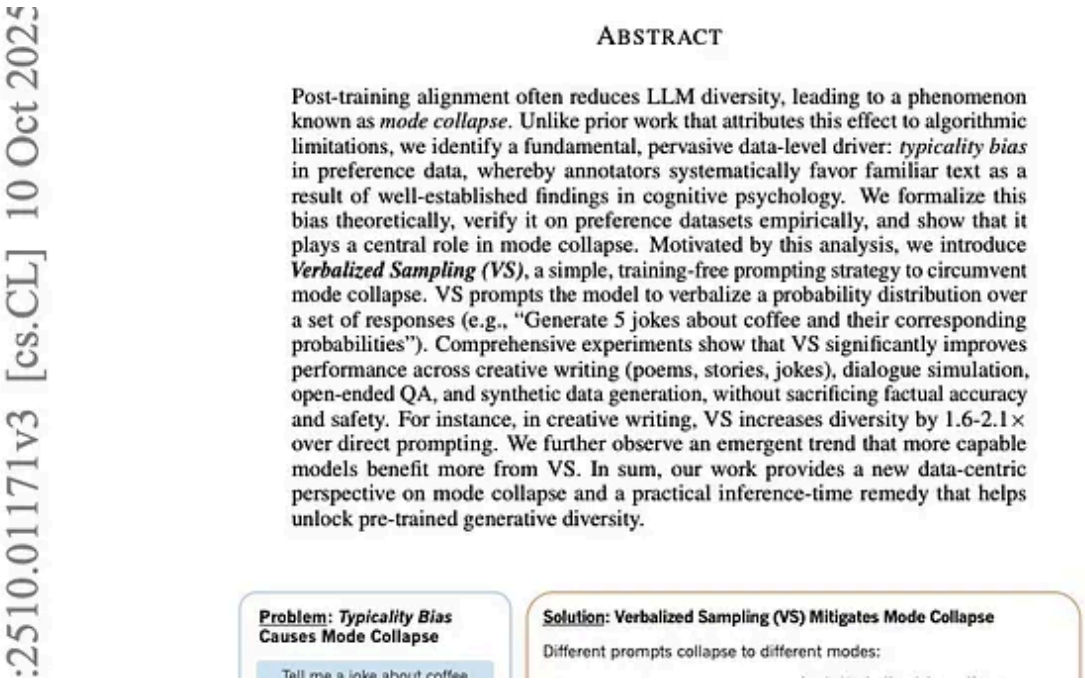


In Write A Catalyst by Dr. Patricia Schmidt

As a Neuroscientist, I Quit These 5 Morning Habits That Destroy Your Brain

Most people do #1 within 10 minutes of waking (and it sabotages your entire day)

Jan 14 26K 449



:2510.01171v3 [cs.CL] 10 Oct 2025

ABSTRACT

Post-training alignment often reduces LLM diversity, leading to a phenomenon known as *mode collapse*. Unlike prior work that attributes this effect to algorithmic limitations, we identify a fundamental, pervasive data-level driver: *typicality bias* in preference data, whereby annotators systematically favor familiar text as a result of well-established findings in cognitive psychology. We formalize this bias theoretically, verify it on preference datasets empirically, and show that it plays a central role in mode collapse. Motivated by this analysis, we introduce ***Verbalized Sampling (VS)***, a simple, training-free prompting strategy to circumvent mode collapse. VS prompts the model to verbalize a probability distribution over a set of responses (e.g., "Generate 5 jokes about coffee and their corresponding probabilities"). Comprehensive experiments show that VS significantly improves performance across creative writing (poems, stories, jokes), dialogue simulation, open-ended QA, and synthetic data generation, without sacrificing factual accuracy and safety. For instance, in creative writing, VS increases diversity by 1.6-2.1× over direct prompting. We further observe an emergent trend that more capable models benefit more from VS. In sum, our work provides a new data-centric perspective on mode collapse and a practical inference-time remedy that helps unlock pre-trained generative diversity.

Problem: *Typicality Bias* Causes Mode Collapse

Solution: Verbalized Sampling (VS) Mitigates Mode Collapse

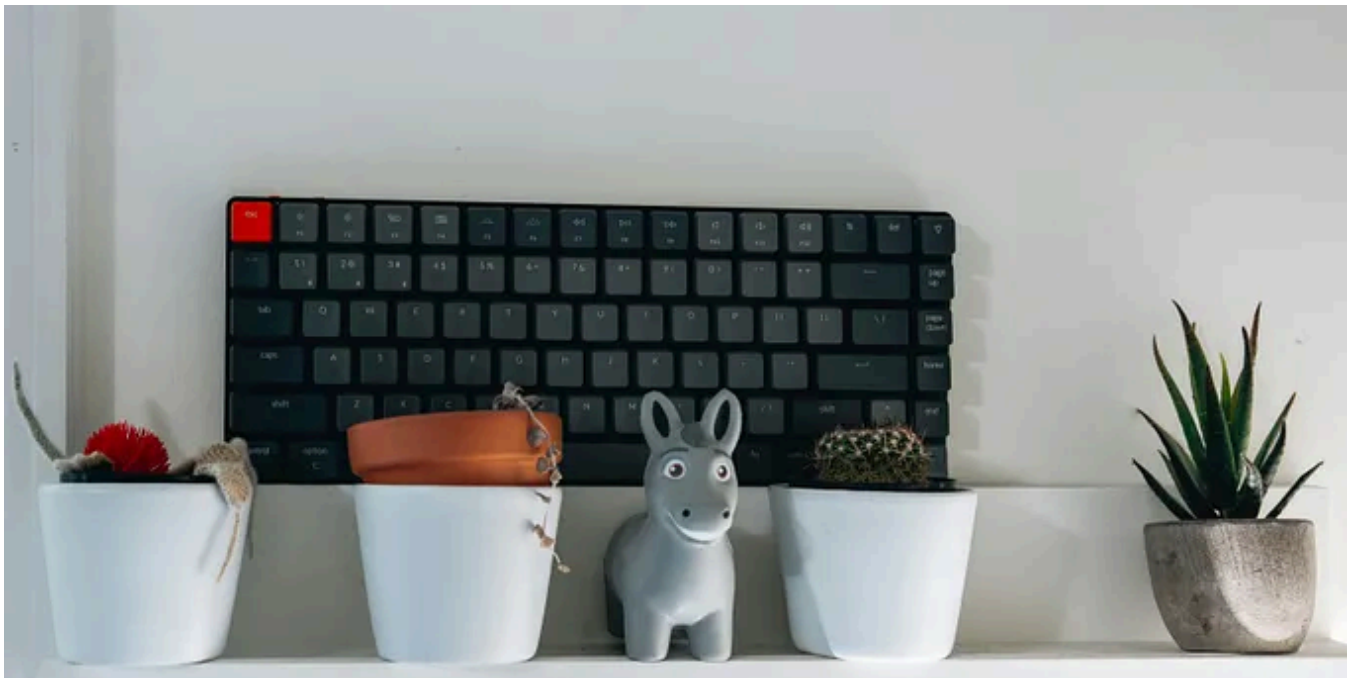
Different prompts collapse to different modes:

In Generative AI by Adham Khaled

Stanford Just Killed Prompt Engineering With 8 Words (And I Can't Believe It Worked)

ChatGPT keeps giving you the same boring response? This new technique unlocks 2× more creativity from ANY AI model—no training required...

Oct 20, 2025 23K 608

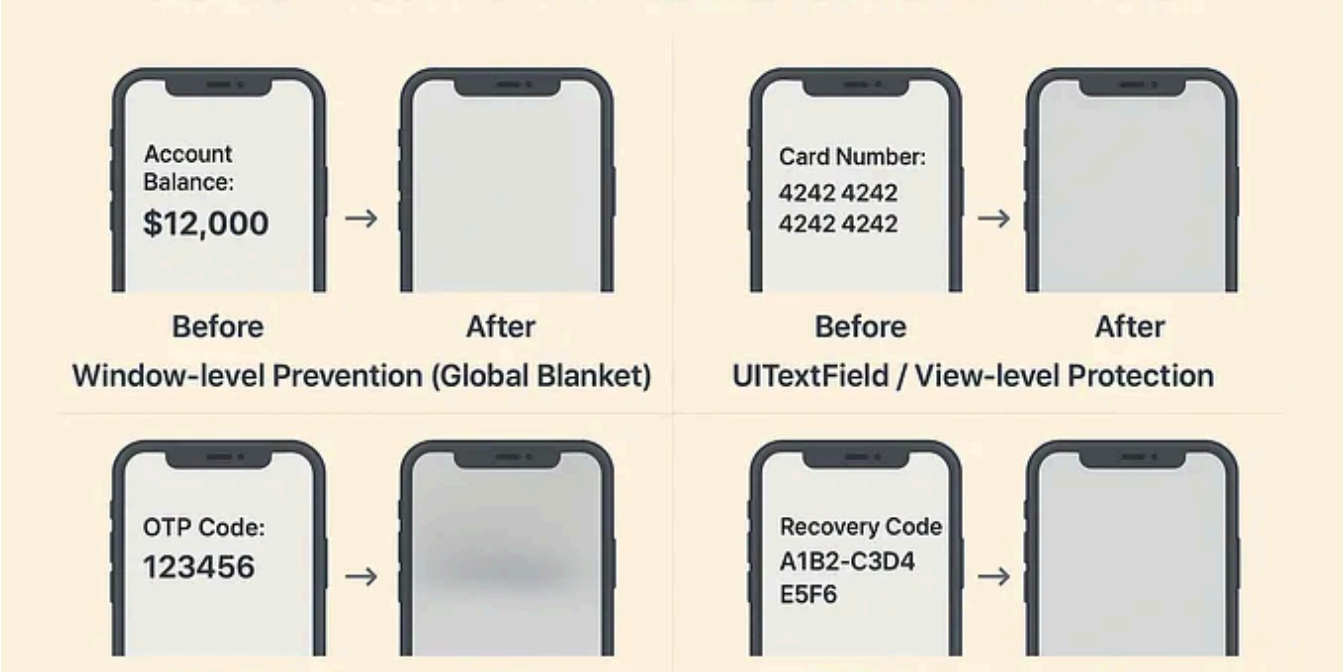


Hemanth KV

React Native Retry Logic—Making Your App Network Resilient

When you're working with APIs in React Native, network requests can fail due to temporary issues like:

Nov 6, 2025 60



Neeshu Kumar

Screenshot Prevention in iOS: Advanced Techniques Inspired by Banking & Authenticator Apps

If you've ever used a banking app or an authenticator app, you've probably noticed a unique security feature: you can't take a screenshot...

Sep 17, 2025 18 2