{callstack}

REACT NATIVE PERFORMANCE OPTIMIZATION WITH DMAIC

A Practical Guide and Case Study in Using Data-Driven Methodology to Obtain Measurable Performance Gains

Is your application struggling with sluggish performance, frustrating users and impacting your bottom line? If so, you downloaded just the right resource.

This guide includes a real-world case study demonstrating how a structured, data-driven approach can transform a slow, unresponsive app into a smooth, high-performing experience. We'll explore the DMAIC methodology—a powerful framework for tackling performance bottlenecks in React Native applications—and showcase its effectiveness through a real-life case study, providing a clear understanding of how it can be applied to your own projects.

To give you a concrete idea of what you can achieve with DMAIC, consider these results:

- App start time slashed from a cumbersome 48.5 seconds to a swift 16.8 seconds.
- Message sending time—a critical user interaction—reduced by nearly 83%, from 14.5 seconds to just 2.5 seconds.
- Report screen load time saw a significant decrease from 15.9 seconds to 8.6 seconds.
- Search screen load times optimized from 11.8 seconds to a mere 2.6 seconds.

Now let's dive into the details of how this transformation unfolded.

THE PERFORMANCE STRUGGLE

A financial application serving millions of users across web, mobile, and desktop was experiencing performance bottlenecks that impacted user experience. This was particularly noticeable in core workflows, such as sending messages within the app - a central feature due to the product's chat-based nature.

As the client embarked on a transformation to a new version of their application, the need for performance optimization became apparent - and we joined forces to make it happen. Our approach was grounded in a structured and data-driven methodology of DMAIC.

We focused on a specific core metric: the time it took to send a message within the app. On accounts with a large amount of data, this action could take nearly 15 seconds, a clear indicator of a significant performance bottleneck. This was our starting point, and we knew we could do better.

Before we share the details of this collaboration, though, let us explain what makes DMAIC particularly fit for performance optimization like this one.

DMAIC: A STRUCTURED APPROACH TO REACT NATIVE PERFORMANCE OPTIMIZATION

At Callstack, we believe in a structured, data-driven approach to performance optimization. That's why we rely on the **DMAIC framework** (the abbreviation stands for Define, Measure, Analyze, Improve, and Control) to consistently improve application performance in a measurable and manageable way. This methodology ensures that both business needs and user experience are prioritized throughout the optimization process.

Before we dive into how we applied DMAIC to the financial app mentioned above, let's explore the general principles of each phase:

- **Define:** This phase focuses on understanding the core of the application. It involves clarifying business objectives, user needs, and specific areas that matter most for the end-user experience. Key activities include defining industry-specific requirements (e.g., an e-commerce app might prioritize the speed of browsing products, whereas a social media app may focus on minimizing the time it takes for the user's feed to load), identifying critical screens and interactions, as well as establishing clear and measurable metrics (e.g., going for "the product list screen should load in under 150ms on Android devices" instead of vague goals like "app loads fast") that will guide the optimization process.

- **Measure:** Once we know what needs to be optimized, the next step is to measure the current performance of the application. This involves collecting reliable baseline data to understand how the app is performing against the desired results. Key aspects of this phase include ensuring the reliability of measurements, using iterative testing with Maesto, Detox, or Appium to obtain more reliable results, and leveraging various tools (APM tools, Flashlight, React Native Performance) to gather baseline metrics.

- **Analyze:** With reliable data in hand, it's time to dig deeper and identify where the performance issues lie. The Analyze phase focuses on finding the root causes of inefficiencies and determining which areas of the app need improvement. This involves profiling the application with tools like Android Studio Profiler, Xcode Instruments, React DevTools, and Hermes Profiling to identify bottlenecks and produce a list of actionable items—tasks that developers can tackle to address performance bottlenecks, whether they involve code optimizations or architectural changes.

- **Improve:** Once the root causes are identified, it's time for action. The Improve phase involves implementing solutions to address the issues found during the Analyze phase. This can include techniques such as memoization, lazy loading, code splitting, deferred execution, efficient navigation, and general code optimization. This phase is iterative, with re-measurement after each optimization to ensure the desired impact.

- **Control:** Performance optimization isn't a one-time task; it's an ongoing process. After implementing optimizations, we move to the Control phase to ensure that the improvements are sustained and that new regressions don't occur. Key strategies include performance monitoring tools and automated performance testing integrated into the CI/CD pipeline.

Fig.1. DMAIC performance optimization process

DMAIC IN ACTION: A REAL-LIFE CASE STUDY

Now, let's see how we applied the **DMAIC framework** to the specific challenges faced by our client.

DEFINE

The first step involved a thorough understanding of the client's business needs and core workflows. We engaged in extensive discussions with their team to identify the most crucial user tasks and establish clear, measurable metrics. This phase culminated in **establishing the target for sending a message at a mere 100 milliseconds.**

MEASURE

With clear objectives defined, the next phase focused on establishing baseline values for the identified metrics. Through a combination of manual testing, automated workflows using Maestro, and custom-built DevTools, we accurately measured the application's performance.

This revealed **the stark reality of the 14.5-second average message sending time**, along with baseline data for other critical metrics like app start time, report screen load time, and search screen load time (see the before and after comparison below).

ANALYZE

With baseline values in hand, we delved into a comprehensive analysis of the performance bottlenecks. We adopted a bottom-up approach, starting with device-level profiling using Android Studio Profiler and Flashlight. We then focused on the JavaScript thread using Chrome Profiler and React Native Release Profiler.

This detailed analysis pinpointed several critical issues, including **inefficiencies in string comparison within the JavaScript engine and a counterproductive caching mechanism.**

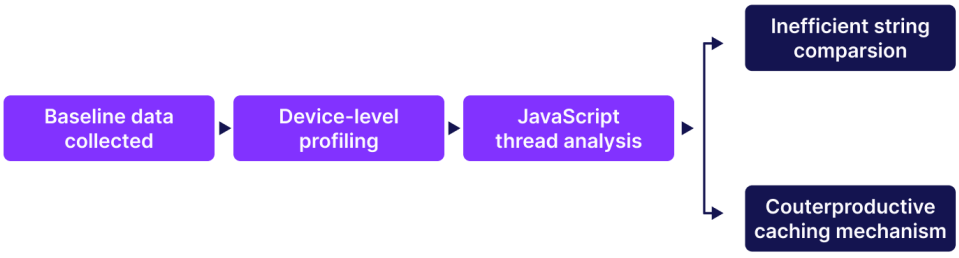

Fig.2. Comprehensive performance bottleneck analysis

IMPROVE

Armed with a clear understanding of the root causes, we implemented targeted solutions. These included **optimizing the string comparison in the JavaScript engine, removing the problematic caching mechanism, and other code enhancements.**

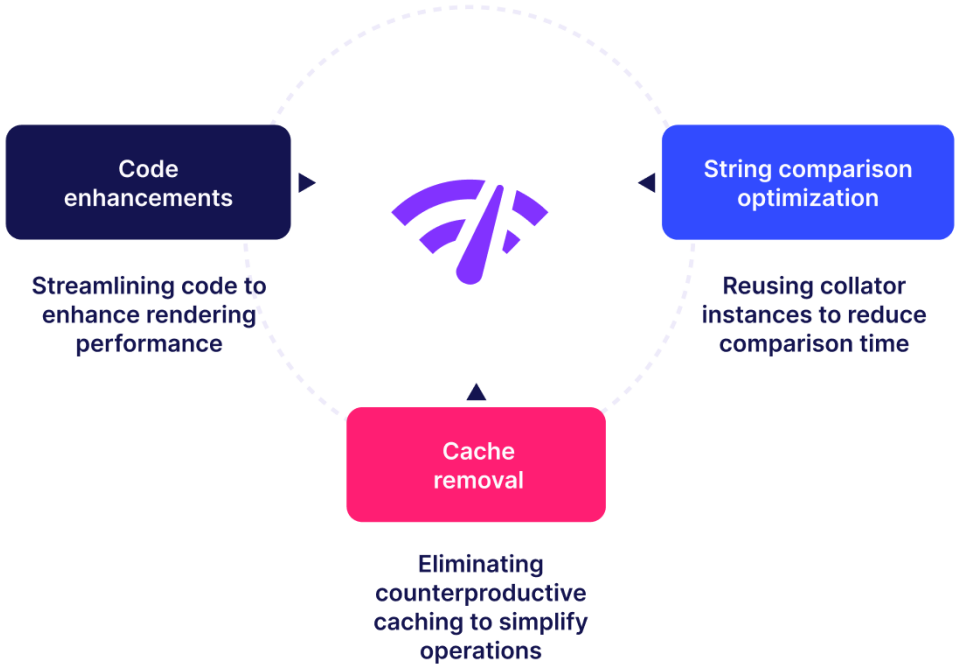

Fig.3. Targeted solutions aiming at performance optimization

To give you a better idea of what improvements we made and how they impacted the app, here's a breakdown of three changes:

Replace `localCompare` calls with static `Intl.Collator`

Before:

```
> reports.sort((a, b) => (a.name && b.name ? a.name.
toLowerCase().localeCompare(b.name.toLowerCase()) :
0));
```

After:

```
// Custom localeCompare function

const DEFAULT_LOCALE = 'en';

const COLLATOR_OPTIONS: Intl.CollatorOptions = {usage:
'sort', sensitivity: 'base'};

let collator = new Intl.Collator(DEFAULT_LOCALE,
COLLATOR_OPTIONS);

function localeCompare(a: string, b: string) {
    return collator.compare(a, b);
}

export default localeCompare;

// Usage
reports.sort((a, b) => (a.name && b.name ?
localeCompare(a.name, b.name) : 0));
```

Improvement: 19% shorter sorting time

Optimization of filtering functions

Before:

```
reports.filter((a, b) => {

            const compareDates = a.created && b.created
? compareStringDates(b.created, a.created) : 0;

            return compareDates || compareDisplayNames;

        });
```

After:

```
reports.filter((a, b) => {

            const compareDates = a.created && b.created
? compareStringDates(b.created, a.created) : 0;

            if (compareDates) {

                return compareDates;

            }

            const compareDisplayNames = a.name &&
b.name ? localeCompare(a.name, b.name) : 0;

            return compareDisplayNames;

});
```

Improvement: 40% faster filters

Counderproductive cache removal

Before:

```
const itemIDsCache = new Map<string, string[]>()

function setCacheWithLimit(cache: Map<string,
string[]>, key: string, value: string[]): void {
      cache.set(key, value)

}

let hasInitialActions = false
const lastItemActions: Record<string, unknown> = {}

function getOrderedItemIDs(

      currentItemID: string | null,
      allItems: Record<string, Item>,
      featureFlags: FeatureFlag[],
      policies: Record<string, PolicyType>,
      priorityMode: PriorityMode,
      allItemActions: Collection<ItemAction[]>,
      itemViolations: Collection<ItemViolation[]>,
      currentPolicyID = '',
      policyMemberIDs: number[] = [],
): string[] {

      const currentActions = allItemActions?.
[currentItemID ?? '']

      let actionCount = currentActions?.length ?? 0
      actionCount = Math.max(actionCount, 1)

      const cachedItemsKey = JSON.stringify(

        [currentItemID, allItems, featureFlags,
policies, priorityMode, actionCount, currentPolicyID,
policyMemberIDs],

        (key, value) => (['someLargeProperty',
'someOtherField'].includes(key) ? undefined : value),

      )

      const cachedIDs = itemIDsCache.get(cachedItemsKey)
      if (cachedIDs && hasInitialActions) {

        return cachedIDs
      }
```

```
hasInitialActions = Object.values(lastItemActions).
length > 0


const isInSpecialMode = priorityMode === 'SPECIAL_MODE'
const isInDefaultMode = !isInSpecialMode

const allItemsArray = Object.values(allItems)

const pinnedItems = allItemsArray.filter((item) => item.
isPinned)
const draftItems = allItemsArray.filter((item) => !item.
isArchived && false)
const nonArchivedItems = allItemsArray.filter((item) =>
!item.isArchived)
const archivedItems = allItemsArray.filter((item) =>
item.isArchived)

pinnedItems.sort((a, b) => a.itemID.localeCompare(b.
itemID))
draftItems.sort((a, b) => a.itemID.localeCompare(b.
itemID))
nonArchivedItems.sort((a, b) => a.itemID.
localeCompare(b.itemID))
archivedItems.sort((a, b) => a.itemID.localeCompare(b.
itemID))

const orderedItemIDs = [

      ...pinnedItems,
      ...draftItems,
      ...nonArchivedItems,
      ...archivedItems,
].map((item) => item.itemID)

setCacheWithLimit(itemIDsCache, cachedItemsKey,
orderedItemIDs)

return orderedItemIDs
```

After:

```
function getOrderedItemIDs(

      currentItemID: string | null,
      allItems: Record<string, Item>,
      featureFlags: FeatureFlag[],
      policies: Record<string, PolicyType>,
      priorityMode: PriorityMode,
      allItemActions: Collection<ItemAction[]>,
      itemViolations: Collection<ItemViolation[]>,
      currentPolicyID = '',
      policyMemberIDs: number[] = [],

): string[] {

const isInSpecialMode = priorityMode === 'SPECIAL_MODE'
const isInDefaultMode = !isInSpecialMode

const allItemsArray = Object.values(allItems)

const pinnedItems = allItemsArray.filter((item) => item.
isPinned)
const draftItems = allItemsArray.filter((item) => !item.
isArchived && false)
const nonArchivedItems = allItemsArray.filter((item) =>
!item.isArchived)
const archivedItems = allItemsArray.filter((item) =>
item.isArchived)

pinnedItems.sort((a, b) => a.itemID.localeCompare(b.
itemID))
draftItems.sort((a, b) => a.itemID.localeCompare(b.
itemID))
nonArchivedItems.sort((a, b) => a.itemID.
localeCompare(b.itemID))
archivedItems.sort((a, b) => a.itemID.localeCompare(b.
itemID))

const orderedItemIDs = [
      ...pinnedItems,
      ...draftItems,
      ...nonArchivedItems,
      ...archivedItems,
].map((item) => item.itemID)

return orderedItemIDs

}
```

Improvement: 10% faster screen load time

CONTROL

The final step focused on ensuring that the performance gains were achieved and maintained through monitoring selected metrics. We integrated Reassure into the testing pipeline for automated performance testing of React components and regular functions, along with Grafana for monitoring. This step was crucial to **prevent regressions and to establish a long-term control policy.**

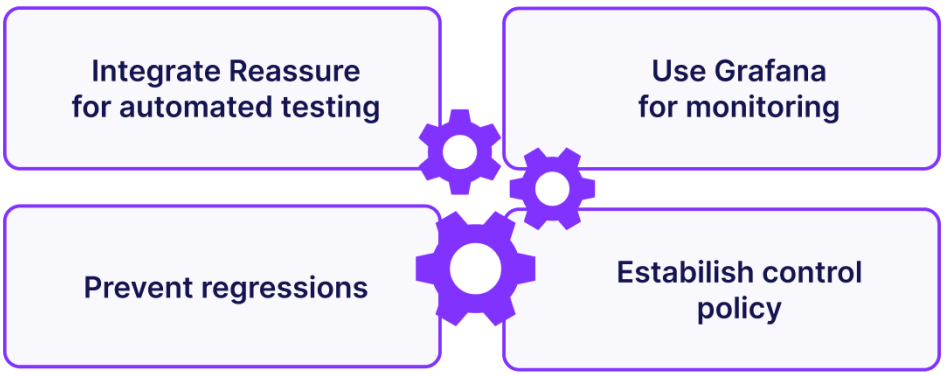

Fig.4. Achieving long-term performance improvement

PERFORMANCE OPTIMIZATION IN NUMBERS

Our analysis revealed a significant bottleneck in the application's JavaScript engine's string comparison process. The default implementation was causing substantial delays, **with string comparisons taking up to 27 seconds on average.** By implementing a solution that reused collator instances, **we were able to reduce this to approximately 5 seconds.** This single optimization had a transformative impact on the user experience.

Another important finding was a caching mechanism that was meant to improve performance but, over time, became a bottleneck. Removing this cache boosted performance and reduced complexity.

The most significant result was the dramatic reduction in the message sending time. **From the initial 14.5-second average, we were able to bring it down to approximately 2.5 seconds.** The almost 83% slash in message sending time was not only a testament to the effectiveness of our approach but also a significant enhancement to the overall user experience. The reduction in time was immediately noticeable to the end-users, making the application feel significantly more responsive and performant.

Metric	Before	After
App start time	48.5s	16.8s
Report screen load time	15.9s	8.6s
Message sending time	14.5s	2.5s
Search screen load time	11.8s	2.6s

Fig.5. Comparison of selected metrics before and after DMAIC-driven performance optimization

BUSINESS IMPACT OF PERFORMANCE GAINS

The impact of these performance gains was substantial:

- **Enhanced User Experience:** The application became significantly more responsive, leading to increased user satisfaction and engagement.
- **Improved efficiency:** The reduction in load times and message sending times streamlined core workflows, making users more productive.
- **Empowered Development Team:** By implementing automated performance testing and monitoring, we empowered the client's team to proactively manage performance and prevent future regressions.
- **Open Source contribution:** Our contributions to Reassure helped the broader React Native community, reinforcing the client's position as a technology leader.

CONCLUSION

This case study demonstrates the power of a focused, methodical, and data-driven approach to performance optimization. By using the DMAIC framework, we were able to transform a complex performance challenge into a resounding success. The results speak for themselves: users now enjoy a faster, smoother, and more responsive application.

{callstack}

ABOUT CALLSTACK

We are more than just optimizers; we are your partners in building future-proof applications. As Meta partners and React Native core contributors, we bring unparalleled expertise to every project.

Our deep understanding of the React ecosystem, combined with a data-driven mindset, allows us to address performance challenges at their core, delivering measurable improvements that last. Whether it's faster load times, smoother animations, or scalability under pressure, we have the skills, tools, and vision to keep your app ahead of the curve.

If you're facing similar performance challenges, contact us for a personalized consultation to assess your needs.