MobileConfig: Remote Configuration Management for Mobile Apps at Hyperscale

Matt Guo[1], Soteris Demetriou[2], Joey Yang[1], Michael Leighton[1], Diedi Hu[1], Tong Bao[1], Amit Adhikari[1], Thawan Kooburat[1], Annie Kim[1], and Chunqiang Tang[1]

[1] Meta Platforms
[2] Imperial College London

Abstract

While software configuration management is a ubiquitous practice in the industry and has been extensively studied, prior research has focused solely on desktop or server applications. This paper presents MobileConfig, perhaps the world's largest configuration management system for mobile apps. It has been in production since 2015 and manages apps running on billions of devices, including Facebook, Instagram, Messenger, and AR/VR/glasses apps. Every day, Meta's developers make a staggering number of live configuration changes, often in the thousands, to remotely control mobile apps, driving them to change runtime behaviors without requiring app code updates. These configuration changes serve diverse purposes such as A/B testing, feature rollout, and app personalization. We discuss how MobileConfig addresses several challenges unique to mobile environments, including (1) the lack of data consistency models that can simultaneously ensure both fast app startup and configuration data freshness; (2) the risk of misconfiguration impacting billions of app users; and (3) the proliferation of mobile client SDKs needed to support diverse mobile platforms, programming languages, and configuration use cases.

1 Introduction

At Meta, we develop tens of mobile apps used by billions of people. Each year, more than 1,000 developers contribute to the codebase of our most popular app. Each week, the app is updated with over 300 code changes, and a new version is released to the app store. This fast-paced development and release of the app, especially when it is collectively done by a large number of developers, pose significant challenges to the reliability of the app. Unlike server-side software, where bug fixes can be deployed instantly under our control, mobile apps lack a transparent method to upgrade from a buggy version to a newer one without user involvement.

Contributions: Chunqiang and Thawan initiated the MobileConfig project in 2014. In terms of paper writing, Chunqiang and Soteris drafted the paper and contributed equally. In terms of coding, Thawan led the project's development from 2014 to 2017, succeeded by Matt until 2023. All other co-authors also made major contributions to the project's development.

We work around the constraint of having no control over app upgrades by providing the ability to remotely modify an app's configuration (*config* for short) and trigger it to change behaviors without a code upgrade. Consider the example of an experimental app feature. Initially, it is gated by a remote config that is enabled for only 0.01% of users. If a bug is detected, the feature can be instantly disabled via a remote config change without user involvement, ensuring that the bug will not be exposed to any user. Besides feature gating, remote configs enable rich functions (§2.1), such as A/B testing [20–22, 39] and personalized user experiences.

Remote configs are extensively used in all our apps, as evidenced by the fact that the frequency of config commits is about 15 times that of code commits. In this paper, we present MobileConfig, our configuration management system for mobile apps, and share our experience in addressing several challenges unique to mobile environments.

Slow app startup. When a developer modifies the value of a remote config on the server side, she typically wants app devices to fetch and apply the new config value soon. For instance, if the config is used to run an A/B test, the developer can obtain useful A/B test results only after a sufficient number of devices have applied the new value. For this reason, Google's Firebase RemoteConfig [16], one of the most widely used mobile configuration systems, suggests that,

> "*If you are loading values for an A/B testing experiment, this strategy (of showing a loading screen to block the app startup until it finishes fetching all the latest configs from the server side) is very strongly recommended [17].*"

Unfortunately, this simple strategy of blocking app startup increases startup time, an important app quality metric [4, 11, 19, 27, 30, 37, 41]. Our evaluation shows that this approach would unacceptably add 2,499 ms to the startup time of our largest app. To put this into perspective, our app performance team often invests months in optimizing an app to reduce its startup time by just tens of milliseconds, and the effort is considered worthwhile as it impacts the engagement of billions of users.

Slow startup plays an outsized role in users' negative perception of an app and discourages users from opening the app,

especially for short interactions such as typing a brief message on a social-network app. In the industry, AppDynamics reported that 60% of users abandon an app after their first usage if they encounter performance problems [5]. At Meta, based on extensive user studies, the product teams set the priority that "*startup is the single most important factor for app performance, and anything not absolutely necessary will be ruthlessly removed from the startup path.*"

At startup, an app wants the latest configs, but fetching them from the server delays startup. This tension has not been addressed by existing data consistency models, as they do not take advantage of the unique characteristics of configs, specifically the timing and granularity of config consumption.

To resolve this tension, we propose *progressive consistency*. Our key insight is that, out of our app's as many as 4,300 configs, only a small fraction is accessed during the app's early startup phase. App startup can proceed by consuming those configs' locally cached values, even if they are stale. In parallel, the app asynchronously fetches updates for all configs from the server. Once the updates arrive, it switches to using the new values for the vast majority of configs that have not yet been accessed. Our production data shows that progressive consistency achieves 99.7% config data freshness across billions of devices without slowing down app startups.

Despite mixed consumption of old and new configs, progressive consistency ensures app correctness by guaranteeing repeatable reads [7], monotonic reads [40], and intra-config consistency. Note that, due to the nature of config usage, it is unnecessary to enforce cross-config consistency (§3.3). To mitigate the downside of consuming some stale configs, MobileConfig offers an emergency-push mechanism that can swiftly purge harmful stale configs from app devices. Finally, it can cap config staleness below an app-specified threshold.

Config error prevention. Every day, our app developers make thousands of live config changes in production, with each change carrying the risk of causing app malfunctions. Moreover, recovering from a config error on mobile devices is much harder than that on servers because unlike servers, we have no access to users' mobile devices to directly rectify the error. MobileConfig uses a defense-in-depth approach to prevent config errors. This includes enforcing compile-time validation, conducting multi-stage canary tests [15], and continuously comparing configs on billions of devices with their values on the server to catch inconsistency.

One surprising finding is that traditional small-scale canary tests are ineffective for mobile apps. This differs from not only the general industry practice [12] but also our own experience in datacenter environments where small-scale canary tests are effective. We find that one cause of this difference is the heterogeneity of mobile environments, including many wild device types and legacy OS versions. As a result, many bugs appear only on a small fraction of user devices, making them hard to detect through small-scale canary tests. Another cause is that, out of a large app's many features, only a small fraction of users may use a specific feature during the canary time window and trigger the bug. Because of these difficulties, MobileConfig uses unconventionally large canary tests.

Proliferation of mobile SDK. MobileConfig supports apps on various platforms (Android, iOS, Windows, Linux, MacOS, and custom OSes for AR/VR/glasses/display) and multiple programming languages (Java, Objective-C, Kotlin, Swift, C++, JavaScript, and ReactNative). The expansion of platforms and languages would naturally lead to an increase in the number of mobile Software Development Kits (SDKs). Moreover, historically, different config use cases such as A/B testing and personalization were supported by separate SDKs and backend systems, leading to further SDK proliferation.

Two key insights help eliminate SDK proliferation. First, to support diverse config use cases at the scale of billions of devices, it is far easier to implement sophisticated capabilities and deploy code changes on the server side than on the mobile client side. On the client side, we simplify the SDK by offering a uniform config API and data schema, agnostic of various config use cases. On the server side, we use a translation layer to dynamically map mobile config parameters to various backends that support different use cases. By introducing this one level of remapping on the server side, a single client SDK can seamlessly work with various backends.

Second, to support diverse platforms and languages, we use a proper mix of cross-platform C++ code and platform-specific code. While C++ is known for cross-platform development, we have discovered that the overhead of using Java Native Interface (JNI) to bridge C++ is too high for latency-sensitive config-read operations. Consequently, on Android, we employ native Java code for these operations, while employing cross-platform C++ code for all other operations.

Contributions. We summarize our contributions below.

- To our knowledge, this is the first systematic study on configuration management for mobile apps. We compare design alternatives and identify pitfalls in some widely used mobile config systems. We also report lessons learned from operating MobileConfig on billions of devices in the wild.
- We demonstrate the practicality of employing remote configs to push the limits of agile app development—our developers make thousands of live config changes daily. We hope that our experience will inspire others.
- We propose *progressive consistency* to meet mobile apps' requirements for both fast app startup and fresh config data. This is not possible with existing consistency models.
- We prevent config errors using a defense-in-depth approach. In particular, we demonstrate that traditional small-scale canary tests are ineffective for mobile apps.
- We avoid SDK proliferation by using cross-platform code and server-side config parameter remapping.

2 Config Usage

In this section, we use production data and examples to demonstrate the usage of configs and the challenges that MobileConfig needs to address. Figure 1 shows an example of how an app uses MobileConfig. In this example, `ButtonCfg` is a *config*, and `ButtonCfg.color` is a *parameter*. The new code path for the experimental "*Button*" feature is gated behind the `ButtonCfg.isEnabled` parameter, which is personalized and can return different values for different users.

```
class ButtonCfg {bool isEnabled; String color; int size;}
class MusicCfg {int volume; String list; bool shuffle;}

if(MobileConfig.getBool(ButtonCfg.isEnabled /* Personalized */)){
    // New code path for the experimental "Button" feature.
    color = MobileConfig.getString(ButtonCfg.color);
    ...
} else {
    // Old code path without the experimental "Button" feature.
    ...
}
```

Figure 1: An app uses a simple API to access remote configs.

2.1 Config Use Cases

Remote configs enable many powerful use cases, as illustrated in the examples below.

- Developers can update a config to incrementally enable an experimental app feature, starting with our employees and gradually expanding to the general population.
- Developers can use a config to set up an A/B test on different users to assess the impact of a new product feature on key business metrics.
- Configs can be used along with machine learning (ML) to personalize user experience. For example, when a user login fails, if ML predicts that the user is unlikely to succeed with password retries, it sends a one-time passcode via SMS to the user. We use a config to store the ML-personalized per-user login retry setting.
- Some parameters that control an app's behavior, such as the amount of data to prefetch from the server, depend on the execution context, such as battery level and network performance. We apply contextual Bayesian optimization to tune these parameters and manage them via a config.
- When datacenters face capacity shortages, specific config changes can be promptly distributed to mobile devices to disable less-essential app features, accordingly alleviating the load they impose on datacenter backend services [25].

Backend	Feature Rollout	A/B Testing	Mutable Parameters	700+ Custom Functions	Dev
Parameters%	26.0%	40.8%	23.0%	2.7%	7.4%

Table 1: Breakdown of config parameters by backends.

To support diverse use cases, many different config *backend* systems have been developed, as summarized in Table 1. The *feature rollout* backend allows hundreds of teams to independently enable or disable different features for users without interfering with one another. The *A/B testing* backend enables developers to study the effectiveness of product features via A/B testing [21]. With the *mutable parameters* backend, instead of hardcoding constants in code, developers can set them as parameters that can be updated remotely without upgrading the installed app. The *custom-function* framework allows developers to easily introduce a new config backend. More than 700 custom backends have been implemented, mostly for personalization and ML model automation. Finally, the *Dev* backend is for local testing only.

Historically, the team that developed a new config backend for a new config use case must also develop a corresponding mobile SDK. Now, different config backends are all supported by a single mobile SDK provided by MobileConfig. This SDK exposes a uniform config API and data schema that are agnostic to different config backends and use cases. On the server side, we use a translation layer to dynamically map mobile config parameters to different backends (§4.1.2).

2.2 Statistics of Mobile Environments

Next, we report some statistics to motivate the problem.

Very old app versions. Users' infrequent app updates necessitate the reliance on remote configs to change app behavior. Table 2 shows the cumulative age distribution of our largest Android app. The (56, 14%) column means that 14% of the app's installations were released in the past 56 days. Notably, some extremely old app versions remain in active use. For instance, 1% of the app's installations are older than 987 days, with the oldest one dating back almost 7 years (2,499 days).

App age (days)	49	56	63	70	77	217	504	987	2,499
Cumulative distribution	1%	14%	34%	56%	73%	90%	95%	99%	100%

Table 2: Very old versions of our Android app are still in use.

Very old OS versions. Our apps run on thousands of different Android device types, with OS versions spanning over a decade (Table 3). This complex environment makes it hard to prevent config errors just by development-time testing. Hence, MobileConfig relies on large-scale multi-stage canary tests in live production as the last line of defense (§5).

Android Version	4	5	6	7	8	9	10	11	12
Release year	2011	2014	2015	2016	2017	2018	2019	2020	2021
Percent of devices	0.1%	1.2%	2.3%	2.8%	9%	12%	26%	38%	8%

Table 3: Very old versions of Android are still in active use.

Low-end mobile devices. The majority of our app users are on low-end devices. We categorized the Android devices on

which our apps run based on the *year-class* metric, which corresponds to the year when the device would have been considered as a flagship device. For example, Samsung Galaxy S6 was released in 2015 as a flagship smartphone. If a low-end device was released in 2023 but its performance is comparable to Galaxy S6, it would be classified as the 2015 class. Table 4 shows that 75% of the devices are comparable to 2015 or older flagship devices. The combination of low-end devices and many configs require us to heavily optimize MobileConfig for performance and efficiency (§6).

Year Class	2010	2011	2012	2013	2014	2015	2016-2023
Percentage	0.04%	0.32%	3.53%	18%	20%	33%	25%

Table 4: Breakdown of Android devices by Year Class.

Many configs. We measured the number of config parameters in our most popular apps, MM and VH (Table 5). They are available on both Android (MM_a, VH_a) and iOS (MM_i, VH_i). The largest app uses more than 4,300 configs and 26,000 parameters. The existence of a large number of independent configs has made progressive consistency possible.

App Name	MM_a	MM_i	VH_a	VH_i
# Configs	4,344	3,546	3,178	3,050
# Parameters	26,770	18,057	8,563	7,828

Table 5: Config usage for our most popular apps.

Frequent config changes. Configs are updated frequently by many authors. The configs of our largest app are edited by more than 3,000 different authors over the app's lifetime. On an average workday, our developers make more than 2,700 config parameter value changes in production, and introduce more than 110 config schema changes. MobileConfig relies on defense in depth to mitigate the risk of config errors introduced by frequent config changes (§5).

Many languages. To show the usage of languages in our apps, we counted the number of static call sites in each language's source code that read remote configs (Table 6). As expected, Java for Android and Objective-C for iOS are most popular, but call sites in other languages still account for 37%. To support multiple languages and OSes while avoiding the development costs of multiple client SDKs, we have implemented MobileConfig's core functions in a portable cross-platform C++ runtime and exposed them to different languages through language-specific bindings (§4.1.1).

Language	Java	Objective-C	Java Script	Kotlin	React Native	Swift	C++
Call site%	35%	28%	18%	11%	6%	0.6%	0.6%

Table 6: Usage of MobileConfig by different languages.

Parameter types. Table 7 shows the breakdown of config parameter types for MM_a and MM_i. As Boolean dominates, MobileConfig implements special optimizations for it (§6.3).

	Boolean	Integer	String	Double	Other
MM_a	69%	23%	4.9%	2.4%	1.3%
MM_i	66%	22%	4.8%	5.1%	1.5%

Table 7: Breakdown of config parameter types.

3 Agile Development with MobileConfig

This section describes how developers use MobileConfig in an agile development process and how it ensures app correctness while enabling both fast app startup and fresh configs.

3.1 Agile Development Process

To illustrate how MobileConfig enables agile development, we describe the workflow of a developer named Alice working on the experimental "*Button*" feature shown in Figure 1.

After sufficient local testing, Alice includes the code for the Button feature in the app's new release and uploads it to the app store. Although some users quickly install the new release, the new feature is not yet exposed to anyone, as `ButtonCfg.isEnabled` is still set to false for all users.

Alice initiates testing for the new feature, initially enabling `ButtonCfg.isEnabled` for only 0.01% of users. Uncertain about the ideal `ButtonCfg.color` and `ButtonCfg.size` values for the best user experience, she sets up an A/B test and divides the test population into multiple groups, each receiving distinct parameter values. By comparing metrics, like user engagement, across the test groups, Alice identifies optimal parameter values. Importantly, the entire A/B test process is driven by Alice making config changes on the server side using a web interface. These config changes are automatically fetched by mobile devices, determining which users will participate in the A/B test and what parameter values they will get for `ButtonCfg`. As Alice makes remote config changes, it does not require participating users to upgrade their installed app or perform any manual operations.

If the app crashes on an old Android version when using the new feature, Alice remotely disables `ButtonCfg.isEnabled` for those users, without requiring an app upgrade. After fixing the bug, she releases a new version of the app to the app store. While still keeping `ButtonCfg.isEnabled` disabled for the affected users on the old version, Alice enables it for a subset of users on the new version to continue testing.

After months of A/B testing and numerous releases, Alice determines that the new feature consistently harms business metrics rather than enhancing them. She abandons the feature, removing the experimental code and releasing a new version to the app store. Notably, the majority of users were never exposed to the feature from its introduction to its removal.

3.2 Stale Configs Hinder Agile Development

While remote configs are supposed to enable agile development, straightforward solutions are ineffective. To highlight the challenges, we analyze the three solutions provided by Google's Firebase [16], which are summarized in Table 8.

	Fast app startup	Repeatable reads	Intra-config consistency	Fresh configs in normal state	Bound staleness if required	Purge misconfig quickly
Firebase (1)	✗	✓	✓	✓	✓	✗
Firebase (2)	✓	✗	✗	Vast majority	✗	✗
Firebase (3)	✓	✓	✓	✗	✗	✗
MobileConfig	✓	✓	✓	Vast majority	✓	✓

Table 8: Comparison of config frameworks. Firebase [16] supports three config consistency models [17].

In the table, Firebase (1) is Firebase's default approach as described in §1, which blocks app startup to fetch configs. This approach would unacceptably increase our largest app's startup time by 2,499 ms.

Firebase (2) uses cached configs to boot the app while asynchronously fetching the latest configs from the server. Once the new configs arrive, the app immediately switches to using them, without ensuring repeatable reads [7] or intra-config consistency. This approach is unacceptable as altering an app's configs during its live execution can result in user-visible anomalies, such as a sudden change in a button's color.

Firebase (3) also uses cached configs to boot the app and asynchronously fetches the latest configs from the server. However, when the new configs arrive, the app saves them to storage and will not utilize them until the next cold start of the app. This approach increases the staleness of configs and slows down the agile development process described in §3.1. Suppose Alice updates an A/B test parameter in the morning with the intention of collecting and analyzing the testing results in the afternoon; she faces a challenge. Despite many user devices fetching the updated parameter in the morning, they will not apply it until the next cold app restart, which may not occur soon. Even if the user switches out of the app, it may remain paused in the background for hours or even days without termination, preventing the new parameter from taking effect when the app is brought to the foreground again. Consequently, Alice cannot gather enough A/B test results in the afternoon or even the following day, significantly hindering agile development.

At Meta, for mobile apps, config commits occur about 15 times more frequently than code commits. This indicates a highly iterative development process where developers invest a significant portion of their time making config changes, collecting and analyzing test results to inform their next steps. Delays in collecting test results caused by stale configs would thus greatly hinder developer productivity.

Rather than delaying the use of new configs until the next cold restart, a potential improvement for Firebase (3) is to apply new configs when the user switches to other apps and puts the app in the background. However, this may still lead to user-visible anomalies. For instance, if a user is midway through reading a news article in the app and switches to other apps, altering the app's configs in the background could lead to the article's text appearing in a different color upon the user's return, as the color is controlled by a config.

Between Firebase (1)'s drawback of slow app startup and Firebase (3)'s drawback of config staleness, Firebase considers the latter to be a bigger problem and hence *"very strongly recommends"* Firebase (1) for apps using A/B testing [17]. In the literature, while the importance of fast app startup is widely recognized [4, 19, 27, 30, 37, 41], the importance of config freshness is often overlooked. Firebase's recognition of this issue based on its experiences with mobile developers is commendable.

3.3 Progressive Consistency

To enable agile development, progressive consistency solves both the problems of slow app startup and stale configs. During an app's startup, it uses cached configs to unblock the app while asynchronously fetching updates for all configs from the server. Once the updates arrive, the app switches to using the new values for the configs that it has not read yet but will stick to the old values for the configs that it has already read.

Our production data shows that progressive consistency achieves 99.7% config data freshness across billions of devices without slowing down app startup. A key reason for the high config freshness is that most configs are not accessed during an app's early startup phase. To aggressively minimize startup time, any code that is not absolutely necessary during startup, such as the initialization of app features that will not be shown on the first user interaction screen, is postponed to later stages. Since the vast majority of app features will not be initialized on the startup path, the configs used by those features will not be accessed during startup either.

Below, we describe how progressive consistency ensures app correctness and discuss the ease of use of its API.

App correctness. Despite mixed consumption of old and new configs, progressive consistency ensures app correctness by guaranteeing intra-config consistency, monotonic reads [40] across app restarts, and repeatable reads [7] within a user session. A session is the time duration between two cold restarts of the app, or it ends early if the user explicitly logs out of the app. A key difference between MobileConfig and Firebase (2), as summarized in Table 8, lies in MobileConfig's support for intra-config consistency and repeatable reads. If the app consumes a parameter value during a session, subsequent reads of that parameter will retrieve the same value, even if it has already been updated on the server. This prevents unexpected app behavior, such as sudden UI button color changes due to parameter updates. Moreover, MobileConfig guarantees intra-config consistency—if the app reads two parameters of the same config in one user session, such as `ButtonCfg.color` and `ButtonCfg.size`, those parameters' values always come from a single atomic update on the server side.

MobileConfig does not guarantee cross-config consistency as configs are intended to be independent, serving different code modules. If dependencies arise among configs, they should be merged into a single config. While merging could

theoretically result in an excessively large config, this has never occurred in the nine years of MobileConfig's production usage. On average, a config contains only 4.3 parameters. Although we have a design for MobileConfig to support cross-config consistency by maintaining metadata about matching versions of interdependent configs, it remains unimplemented due to the absence of a genuine need for such complexity.

Moreover, the lack of support for cross-config consistency is not merely a workaround to expedite startup in mobile environments but extends to datacenter (DC) environments as well, where startup time is not an important consideration. In contrast to MobileConfig's progressive consistency, our DC configuration management system, Configerator [38], adopts Firebase (1)'s approach, blocking a DC application's startup to fetch configs synchronously, as startup time is less important in DCs. However, Configerator still does not guarantee cross-config consistency. It may push real-time updates for different configs to a DC application's running instances in different orders, as configs are independent. Finally, despite config errors being a primary cause of production outages [38], we do not recall that either Configerator or MobileConfig, with 12 and 9 years of production usage respectively, has experienced outages due to the lack of cross-config consistency.

API simplicity. While implementing progressive consistency, ensuring advanced features do not complicate the developer API is a challenge. A transaction-like API to ensure config consistency seems straightforward but could hinder usability and adoption. Instead, MobileConfig offers a straightforward API to apps (see Figure 1) while handling advanced features internally. For example, when the app calls `MobileConfig.getString(ButtonCfg.color)`, MobileConfig must detect if `ButtonCfg.isEnabled` was already read and return the corresponding version of `ButtonCfg.color` for intra-config consistency. This streamlined API has successfully facilitated the migration of about a dozen legacy config frameworks at Meta to MobileConfig.

4 MobileConfig Design

In this section, we describe the design of MobileConfig and compare it to Firebase [16] and Configerator [38].

4.1 MobileConfig Architecture

MobileConfig's architecture is depicted in Figure 2, which encompasses the client library and the server-side components.

4.1.1 Client-side Library

MobileConfig's client-side runtime library (*runtime* for short) needs to support various platforms and languages. To avoid the development costs of multiple client SDKs, the runtime is implemented in portable cross-platform C++. The API layer exposes the C++ runtime to multiple languages.

The Java API uses JNI to bridge to C++, incurring higher

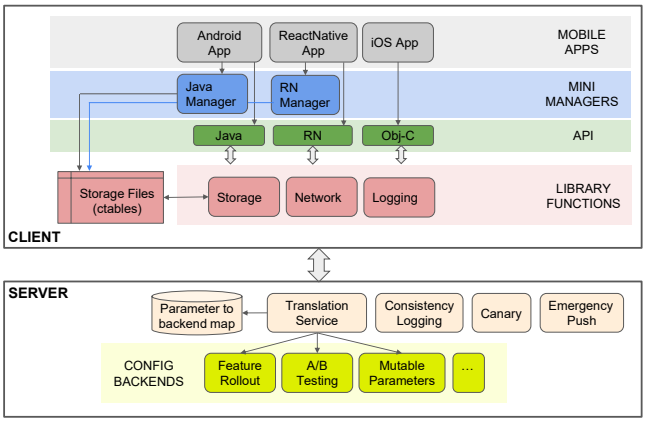

Figure 2: MobileConfig high-level architecture.

overhead. Due to the aggressive optimizations described in §6.2, the C++ API only needs two memory accesses to read a parameter value, which makes the JNI overhead prohibitively high in comparison, especially because config reads can be on the critical path of app execution. Reimplementing the entire runtime in Java would alleviate the problem but at the expense of additional development costs. To strike a balance between code reuse and performance, MobileConfig introduces a *Mini Manager* that implements a minimal read path in Java to efficiently read cached configs. This read path is on the critical path of app performance. All other functions that are not on the critical path, such as fetching configs from the server and updating the cache, are bridged through JNI to C++.

The support for other languages is simpler. The Objective-C API natively interfaces with C++. As ReactNative can cross-compile JavaScript to either Java or Objective-C, MobileConfig offers a JavaScript API that bridges JavaScript functions to either Java or Objective-C functions.

The runtime provides *networking*, *storage*, *exposure logging* and other functions. For *networking*, it efficiently synchronizes config values between client and server (§6.3). For *storage*, it stores configs in a highly optimized read-only binary format for fast access (§6.2). For *exposure logging*, the events of an app's exposure to config parameter values are logged, queued locally, and at opportune moments sent to the server in batches. Finally, the runtime supports operations related to config testing, debugging, canary, monitoring, and quick rectification of erroneous configs (§5).

4.1.2 Server-side Components

On the server side, different backend services support diverse config use cases, such as A/B testing and feature rollout. The *translation service* consults the *parameter-to-backend map* stored in a key-value store to map each config parameter requested by the client to its corresponding backend. This server-side translation drastically simplifies the client runtime, as the client can use a uniform config API and data schema that are agnostic to different config backends and use cases.

The translation service provides an extension point that allows developers to easily add a new custom-function backend, by implementing the following interface: "*generateParamValue (requestContext, userID)*." Based on the *userID* and *requestContext*, a custom backend can personalize parameter values. More than 700 custom backends have been implemented, mostly for different kinds of personalization.

MobileConfig also encompasses tools for detecting, preventing and rectifying config errors. *Consistency logging* samples client and server parameter values to detect config divergence. To prevent config errors, MobileConfig performs multi-stage canary tests when rolling out a config parameter change. Finally, MobileConfig's *emergency push* mechanism can quickly push a corrected config to billions of devices.

4.1.3 Client-server Protocol

Push vs. pull is an important design decision in the client-server protocol. With a push model, the server immediately pushes new data to clients. With a pull model, the client periodically pulls new data from the server in batches.

Table 9 compares push and pull for different systems. Configerator [38], our configuration system for datacenter applications, uses a push model (column 2 in Table 9). Our mobile messaging app uses a push model to deliver chat messages (column 3 in Table 9). No known system uses a push model for mobile configs (column 4 in Table 9). Firebase RemoteConfig [16] uses a pull model for mobile configs (column 5 in Table 9). MobileConfig (column 6 in Table 9) uses a pull model complemented by emergency push, which only happens several times per year to rectify severe config errors.

In datacenter environments, many config use cases require instant delivery for real-time automation, like global load balancing [32]. In contrast, mobile config use cases rarely need instant updates of configs because apps often do not consume them immediately. Apps may not be running or prefer to stick with the old config until the next restart to prevent abrupt user experience changes, such as UI alterations.

To minimize resource consumption, we opt for a pull model. However, in emergency situations such as rectifying config errors, it is necessary to push a config update to mobile devices quickly and restart the app to consume it. Hence, we complement our pull model with occasional emergency push.

4.2 Support for Progressive Consistency

In this section, we present how progressive consistency works during the app lifecycle, and describe the implementation of repeatable reads and intra-config consistency.

4.2.1 Config Consistency during App Lifecycle

When a user installs a new app and logs in for the first time, MobileConfig blocks app startup to fetch the latest config values from the server. This is done to ensure a better user experience than using up to 26,000 unoptimized default parameter values. It is important to note that this blocking does not occur on subsequent app restarts. Moreover, MobileConfig can use *partial fetch* (§6.3) to retrieve only the essential configs and reduce blocking time. If the initial synchronous config fetch fails or times out (e.g., due to no network connectivity), the app proceeds with startup using the default parameter values statically compiled into the app's executable.

During the app's steady-state execution, whenever it enters the foreground, it checks whether H hours have passed since the previous config fetch. If so, it asynchronously fetches configs from the server, and caches configs on disk. We empirically found that H=4 hours strike a good balance between resource consumption and config freshness.

When the app cold-starts next time, it `mmap()` cached binary-format configs into memory, and can immediately read individual config parameters without loading or parsing configs (§6.2). While the app boots normally with cached configs, it immediately issues an asynchronous config-fetch request to the server. Suppose the response comes back with new config values in S seconds (e.g., S=2). The app's future reads to configs that are not read yet will consume the newly fetched config values. Overall, during the first S seconds of app startup, it consumes cached configs; after that, it consumes newly fetched configs. This approach strikes a balance between app start time and config freshness.

To bound config staleness, an app can block its startup to fetch configs if and only if the cached configs are fetched more than T hours ago, where T is configurable per app. If T=48 hours, a person who uses the app daily would never experience the blocking. If T=0, the app always blocks on startup, which is the Firebase-recommended approach. By default, our apps use T=∞ (i.e., never block) because 1) it already provides our apps with the latest version of the vast majority of configs, and 2) our apps can quickly recover from harmful stale configs via emergency push, and hence there is no need to be overly conservative.

After an app executable upgrade (not config update), the app consumes cached configs just like a normal cold start.

		Config Mgmt in Datacenters (push)	Mobile Messaging App (push)	Config mgmt in mobile environ.		
				Push (no real use)	Pull (Firebase)	Pull + Emergency Push (MobileConfig)
# Endpoints		millions	billions	billions		
Endpoint availability		high	low	low		
Message fanout		millions	<100	billions		
Message rate		high	low	medium		
Push notif. reliability		high	medium	medium		
Hardware resources		abundant	scarce	scarce		
Outcome: infra cost relative to requirements		low	low	high ✗	low	low
Reliability	Requirement	high	medium	high		
	Outcome	high	medium	medium ✗	high	high
Delivery Speed	Requirement	fast	fast	usually slow but fast in emergency		
	Outcome	fast	fast	fast	always slow ✗	usually slow but fast in emergency

Table 9: Comparison of push vs. pull.

Some new config parameters might be added to the app's new version and hence do not exist in the local cache. The app can boot using those parameters' default values until the first asynchronous config fetch finishes. Alternatively, the app can block on a *partial fetch* (§6.3) to retrieve a minimal subset of those missing parameters that are important to the app startup.

4.2.2 Repeatable Reads and Intra-config Consistency

Progressive consistency guarantees that 1) during a user session, an app's multiple reads to the same parameter always return the same value, and 2) parameters in the same config are consistent with each other when consumed by the app.

Once every H hours, the MobileConfig runtime fetches the latest config values from the server, and saves configs in a so-called `ctable` file, resulting in a sequence of `ctables` over time: `0.ctable`, `1.ctable`, `2.ctable`, and so on. Old versions are garbage collected when they are no longer used.

On app startup, suppose `0.ctable` exists and it contains the configs in Figure 1. When the app reads `ButtonCfg.color`, the runtime will retrieve it from `0.ctable`. Later, after the runtime fetches new configs and creates a new version, `1.ctable`, the app reads `ButtonCfg.color` again and the runtime will still retrieve it from `0.ctable` (instead of `1.ctable`) in order to ensure repeatable reads. Later, when the app reads parameter `ButtonCfg.size`, the runtime will also retrieve it from `0.ctable` in order to ensure that the app reads `ButtonCfg`'s parameters from the same version.

Later, the app reads `MusicCfg.volume` and the runtime will retrieve it from `1.ctable` since that is the latest cached version. Note that the app reads `ButtonCfg` and `MusicCfg` from different versions of `ctables`. Treating each config independently allows the app to consume each config's latest cached version on its first access to the config.

5 Config Reliability

Multiple factors make it challenging to prevent config errors at scale. First, at Meta, hundreds of developers modify one app's code concurrently and release a new version every week. Second, they also make config changes in live production thousands of times per day. Finally, our apps run on billions of devices in the wild—various OSes and unmaintained old versions, plus thousands of wild Android device types. Hence, we have to rely on defense in depth for reliability.

MobileConfig automatically runs multi-stage canary tests on config changes. As soon as a config change is code-reviewed and accepted, MobileConfig tests the change in production by randomly selecting 0.5% of users as the canary group and another 0.5% as the control group. The canary test spans 30 minutes, during which the MobileConfig runtime uses exposure logging to report the app's consumption of the new parameter values. Once the canary time expires, our tool checks whether the canary parameters are associated with regressions in key metrics. If so, the change is reverted.

After the first-stage canary, MobileConfig initiates a more thorough second-stage canary to catch harder-to-detect config errors. It partitions the entire population into a 50% canary group and a 50% control group. This longer, four-hour canary primarily monitors key metrics such as app crashes.

The second-stage canary (50% of users) is much larger than the common practice [12]. For datacenter applications, we also do not run canary tests at this large scale. However, the mobile environment is diverse, with various device types and older OS versions. Consequently, bugs may only manifest on a small subset of devices, making them hard to detect in small-scale canary tests. Moreover, in a large app, only a fraction, or even a very small fraction, of users may use a specific feature during the canary window, potentially triggering a bug. Due to these challenges, MobileConfig employs unusually large and unusually long canary tests in the second stage.

While more intermediate stages could be added to MobileConfig's multi-stage canary, doing so would further extend the rollout time for config changes, already at 4.5 hours, significantly longer than in datacenter environments. Given the noisy mobile environments, it requires a longer canary time compared to datacenter environments. To reduce the rollout time, we opt for the minimum of two stages. The first stage promptly identifies obvious problems within a small user population, preventing widespread impact, while the second stage, with a large population, catches difficult-to-detect issues, serving as a robust last line of defense.

Emergency push. Severe config errors have to be fixed quickly by emergency push, as the normal process of updating a config can take a long time to reach most devices and even longer for apps to restart and consume the new config.

Emergency push works as follows. When a severe config error is either detected by multi-stage canary or manually reported, the developer can start an emergency push by specifying the target devices to be notified, e.g., based on device model, app version, country, user's spoken language, etc. The server maintains for each config an emergency version number (EVN), which is incremented whenever an emergency push happens to the config. The config's name and latest EVN are pushed to the target devices via our own implementation of an MQTT-based push notification mechanism.

The device compares the received EVN with its local EVN. If the former is higher, the device requests the latest parameter values and an action from the server. The action options are: *do nothing* (new values take effect after next app restart), *force refresh* (the next parameter read will consume the new value), *background restart* (the app will restart when switched to background), and *foreground restart* (the app restarts immediately, even if in foreground). *Foreground restart* is the most disruptive to users and is reserved for severe issues.

Config consistency checking. MobileConfig continuously monitors config consistency between clients and the server. The MobileConfig runtime periodically captures snapshots

of a device's local configs and transmits them to the server. The server retrieves parameter values from config backends, compares them to the client-reported values, and logs results in a database. Numerous health monitoring tools scan the database to detect issues promptly.

6 Performance Optimizations

In this section, we elaborate on several optimizations that allow MobileConfig to operate at scale efficiently.

6.1 Optimizing Strongly Typed Parameters

MobileConfig uses a strongly typed config API to prevent type errors. Given a config schema, our tool generates strongly typed parameter definitions for multiple languages. Apps use the strongly typed API to access parameters, for example, *MobileConfig.getString(ButtonCfg.color)*. By contrast, apps using Firebase [16] access parameters through *untyped* string identifiers, which can result in runtime type errors.

Unfortunately, the benefits of strong types come with a high cost. The symbols of thousands of Java config classes and parameters would inflate an Android app's binary size and slow down app startup. To mitigate this, our compilation tool transparently replaces all Java config class and parameter symbols with encoded integer IDs, which serve as indices to efficiently locate parameter values in `ctable` files at runtime.

6.2 Optimizing Config Storage

Config storage also affects the app startup time. Our evaluation shows that using Firebase [16]'s approach of storing configs in JSON files on Android would unacceptably prolong our largest app's cold start time by 558ms. Hence we heavily optimize config storage for high performance.

Config storage format. To enable fast config reads, MobileConfig uses flatbuffers [18] to encode hierarchical config data into an efficient byte-array representation. For each parameter type, it creates a byte array that contains two sections: 1) the values of parameters of the given type, and 2) metadata for each parameter, e.g., a `loggingEnabled` field indicating whether an exposure event should be logged. The per data-type byte arrays are then concatenated, with header and tail sections added. The header contains the offsets of the per-type subarrays. The tail contains the LoggingIDs for each parameter. Finally, the entire byte array is persisted on disk as the so-called `ctable` file.

Fast parameter read. On app startup, the MobileConfig runtime `mmap()` the `ctable` file so that the app can immediately read specific parameters without loading all configs into memory or parsing configs. On a parameter read operation, the runtime extracts from the encoded 64-bit parameter ID the following metadata: the config's rank among all configs, the parameter's rank, and the parameter's type. Then it uses them as indices to efficiently locate the parameter's value, metadata, and LoggingID in the `ctable` file.

6.3 Optimizing Client-Server Protocol

A straightforward implementation of the protocol described in §4.1.3 would be inefficient. This section describes the optimizations we have made to the baseline protocol.

Partial fetch. Usually, an app asynchronously fetches configs from the server without blocking app startup. However, there are two exceptions: the first login after app installation and the first startup after an app upgrade (§4.2.1). One important insight is that even if occasional synchronous config fetches are necessary, it is unnecessary to fetch all config in a single batch because most configs are not used in the first few seconds of app startup. Hence, it is likely that an asynchronous fetch can finish before those parameters are used.

To minimize the delay, MobileConfig uses *partial fetch*, which only retrieves the minimum subset of configs needed in the early phase of app startup. These parameters are identified through tests in our lab environment by tracing the parameters read during app startup until it is ready for user interaction.

Config schema hash. Apps of different versions use different config schemas and the server does not know the exact list of configs and parameters that a client wants to fetch. Since each app version is associated with a fixed set of configs and parameters, the server only needs to know the client's app version. At compilation time, our tool generates a SHA-256 hash of the list of configs and parameters, and stores it along with the app binary. At runtime, the client sends to the server the SHA-256 hash, and the server consults a key-value store to map the hash back to a list of configs and parameters.

Parameter value hash. Sometimes the server may wastefully send a parameter's latest value to the client while the client's cached version is already up-to-date. To avoid this overhead, the client partitions its configs into sets, and produces a hash for parameter values in each set. The client includes these hashes in its request to the server, which are used by the server to identify and skip unchanged config sets.

Boolean encoding. Most parameters are booleans (Table 7). We reduce the server response size by using two bits to represent a boolean (null/valid and true/false) and then concatenating all boolean bits into an efficient byte array.

7 Evaluation

Unless otherwise noted, all experiments described in this section are conducted in production at hyperscale.

7.1 Usage and Adoption

MobileConfig has been in production since 2015 and has become the only mobile config solution at Meta after consolidating about a dozen different legacy solutions. Currently, it manages tens of apps running on billions of devices. Table 10 summarizes the usage statistics for several apps.

The largest app, MM_a, uses more than 26,000 config pa-

App	MM_a	$MM_{a,L}$	MM_i	VH_a	VH_i	TH_a	TH_i	BA_a	BA_i	WP_a	WP_i
# Configs	4344	46	3546	3178	3050	2780	1177	4680	3254	4381	2864
# Parameters	26770	362	18057	8563	7828	17662	4793	28189	17223	27192	15885
# Projects	5797	92	2437	1607	1642	4650	645	6727	2644	6596	2385
# Teams	754	19	607	336	315	532	185	725	539	718	505
Config fetches per week (billion)	64	8	26	52	38	34	46	0.15	0.2	0.02	0.02

Table 10: Config usage statistics. *Projects* are makefile-like compilation targets. The subscripts a and i mean Android and iOS, respectively, e.g., MM_a and MM_i. The business apps (BA and WP) have less users than the other consumer apps.

rameters and is jointly developed by more than 700 teams. The smallest app, $MM_{a,L}$, is a lightweight version of the MM app, optimized for minimal resource consumption on low-end Android devices. $MM_{a,L}$ uses 362 config parameters, which is about two orders of magnitude smaller than that of MM_a.

Overall, Table 10 shows that MobileConfig is capable of supporting both small and large apps. Moreover, the large number of teams and projects in Table 10 highlights that the agile development process (§3.1) enabled by remote configs can scale to many people jointly working on one large app.

7.2 Impact on App Startup Time

Both our apps and MobileConfig are aggressively optimized for fast startup because it directly impacts user engagement.

7.2.1 Consistency Model's Impact on App Startup Time

Figure 3 shows the average config-fetch time measured in production for different apps. Blocking app startup while fetching configs, as Firebase does [17], would significantly prolong the startup times of our apps by anywhere from 1091ms to 2866ms, causing a detrimental impact on user engagement. To put this into perspective, our app performance team often invests months in optimizing an app to reduce its start time by just tens of milliseconds. The config-fetch time is longer than the network round-trip time because it includes time for the config backends to generate personalized values for as many as 27,000 config parameters. Overall, the long config-fetch times emphasize the importance of progressive consistency's approach of using cached configs to unblock app startup.

MobileConfig performs synchronous config fetches in two cases: the first login after app installation and the initial startup following an app upgrade. In these cases, it employs partial fetch (§6.3) to retrieve only the most important configs. For our popular app, MM_a, compared to a full fetch, partial fetch reduces fetch time from 2,499ms to 1,571ms and decreases transferred data from 401KB to 31KB. The smaller reduction in fetch time is due to the non-linear relationship between config size and config-fetch time. The fetch time includes computation time for the config backends to generate personalized values for many parameters in parallel, which does not decrease linearly as the config data size reduces.

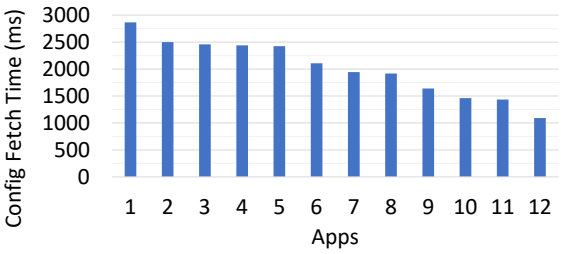

Figure 3: Config-fetch time for different apps.

7.2.2 Config Storage's Impact on App Startup Time

On app startup, the MobileConfig runtime directly `mmap()` a binary-format `ctable` file into memory and immediately read individual config parameters, without the delay of parsing configs or loading all configs into memory. By contrast, Firebase [16] on Android stores configs in JSON files. To do a direct comparison, we modified our VH_a app to use a similar approach to store configs in JSON, and we call it $VH_{a,json}$. On startup, both VH_a and $VH_{a,json}$ use cached configs, and hence the impact of config fetch is excluded from the comparison. We measured app startup time in production. On average, VH_a boots 558ms faster than $VH_{a,json}$. This significant win underscores the importance of optimizing storage format and access method. Unlike $VH_{a,json}$, which parses the JSON file to extract all parameters before accessing even a single one, MobileConfig utilizes `mmap()` to access configs in an optimized binary format. This enables it to selectively page in the specific page containing the needed parameter and read it directly, without being concerned about other parameters.

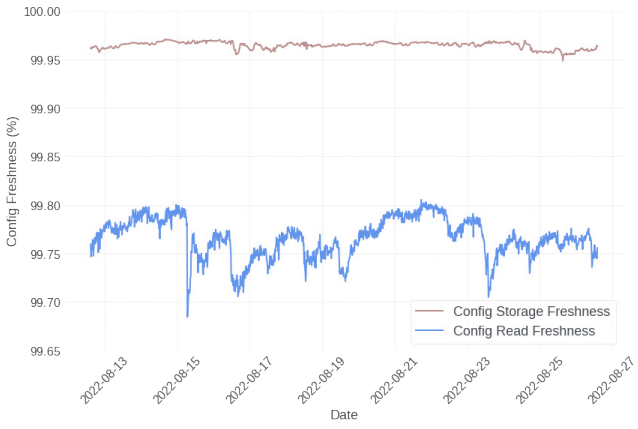

Figure 4: Config freshness for the MM_a and MM_i apps.

7.3 Config Data Freshness

Progressive consistency not only enables fast app startup but also provides apps with highly fresh configs. We use MobileConfig's *config consistency checker* (§5) to measure config freshness in production. Config *read freshness* measures whether a parameter value consumed by an app is identical to the parameter's value on the server side. Config *storage freshness* measures whether a parameter's value stored in a device's cache is identical to the parameter's value on the server side. Read freshness and storage freshness may diverge because even if a client has fetched and stored a parameter's latest value in cache, the app may keep using an older cached version in order to guarantee repeatable reads.

Figure 4 shows config freshness reported by billions of devices for our biggest app *MM*. The P50 values of config read freshness are 99.70% and 99.84% for MM_a and MM_i, respectively. These demonstrate that progressive consistency is able to provide apps with highly fresh configs while enabling fast app startup.

The config storage freshness is close to 100% but never reaches 100% due to how it is calculated. A config storage sample is taken at time T_0 right after a MobileConfig client fetches configs from the server. Later, the sample is compared with the config data on the server side at time T_1. Between T_0 and T_1, some parameter values might have changed on the server side, causing the sample to be considered "not fresh."

7.4 Emergency Push

Emergency push (§5) accelerates the process of purging harmful stale config data from app devices to rectify config errors. We designed an experiment to measure in production how quickly an update on a specific parameter is disseminated and consumed by app devices. We compare different setups: 1) *baseline*—no use of emergency push, 2) *EP w/o restart*—emergency push without app restart, 3) *EP w/ restart*—emergency push plus forced app foreground restart.

The results are shown in Figure 5. The x-axis is the time since the parameter value is updated. The *freshness coverage* metric on the y axis measures the percentage of parameter reads that return the new parameter value out of all app devices' reads to the parameter. The app is set up to read the specific parameter immediately after it boots. Freshness coverage is calculated in a 30-minute moving time window.

By design, "*EP w/ restart*" guarantees near 100% freshness coverage. If a device's app is running at time 0 when the parameter value is updated, soon the app will receive the new parameter value via emergency push and then immediately restart to consume it. If the app is not running at time 0, when it is opened later, it will boot with the cached old parameter value, but will quickly receive the asynchronously fetched new configs and notice that an emergency push has happened. It will immediately restart the app to consume the new parameter value. If we exclude the old parameter value temporarily consumed by the app during the very short period of time between the app's first startup and its immediate restart, by design "*EP w/ restart*" guarantees near 100% freshness coverage. This is shown as the top curve in Figure 5. Note that in this experiment, the top curve is inferred instead of measured in production because we cannot afford to force-restart the app in production just for an experiment, which would cause a disruptive experience to many real users. Local tests on our devices confirm that "*EP w/ restart*" indeed restarts the app in seconds to consume the new parameter value.

The "*baseline*" curve in Figure 5 shows that without emergency push, it takes a long time for freshness coverage to reach a high value. Specifically, after 4 hours, the coverage reaches 26%; after 24 hours, it reaches 85%. The long tail is caused by users who do not use the app for a long time.

"*EP w/o restart*" improves freshness coverage. Specifically, after 4 hours, the coverage reaches 40% (vs. 26% in "*baseline*"); after 24 hours, it reaches 92% (vs. 85% in "*baseline*"). However, the wide gap in freshness coverage between "*EP w/o restart*" and the ideal setup of "*EP w/ restart*" shows that it is insufficient to just quickly push the new parameter value to devices because the app will not consume it until the app's next restart. Therefore, forced app restart, though disruptive, is a necessary step to quickly purge stale configs.

In production, EP is used approximately once per quarter to rectify config errors and almost all those cases use "*EP w/ restart*." In addition, "*EP w/o restart*" is frequently used for Defcon drills [25], as explained in §8.1, for disaster readiness drills rather than handling real production outages.

7.5 Multi-stage Canary Tests

Multi-stage canary tests help catch code or config bugs early. Over a one-month period, MobileConfig conducted 81,014 canaries and caught 15 bugs, all in the second-stage canary. All these bugs slipped through the small-scale first-stage canary, because the regression in app health metrics was too subtle to be reliably detected in very noisy mobile environments. This highlights the difficulty of config error prevention in mobile environments. Despite a large body of research on config error prevention [9, 14, 24, 29, 44, 46, 47, 49], we found that large-scale canary tests in production are still the most robust and widely applicable method.

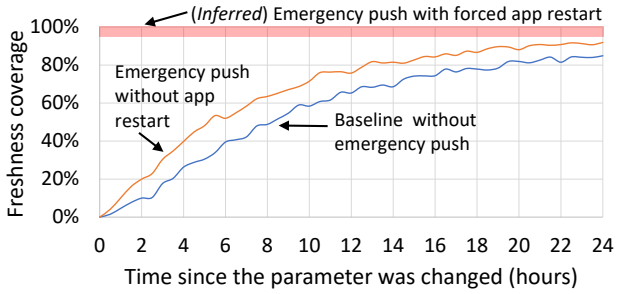

Figure 5: Impact of emergency push on freshness coverage.

App	All OFF		QH OFF		VH OFF		BE OFF		VH+BE OFF		All ON (production version)			Boolean Count (%)
	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Request (KB)	Response (KB)	Total savings (%)	
MM_a	148	880	181	6.4	0.85	698	33	802	0.85	880	33	5.7	96	70
WP_a	145	808	175	6.3	0.82	642	31	738	0.82	808	31	6.7	96	69
MM_i	102	473	125	3.4	0.74	354	24	533	0.75	476	24	3.9	95	66
WP_i	84	374	101	1.5	0.64	287	19	417	0.64	374	19	4.7	95	65

Table 11: Impact of client-server protocol optimizations: Value Hashing (VH), Boolean Encodings (BE) and Query Hashing (QH).

We describe two prevented bugs below to give some intuition on why the second-stage canary caught the bugs but not the first-stage canary. In the first example, a developer used remote config to drive an A/B test that targeted a very small population *P* of Android users. Because MobileConfig's first-stage canary only samples 0.5% of those *P* users, i.e., an even smaller population, it did not catch any problem. The second-stage canary was conducted on 100x more users and identified about 3,000 app crashes that only happened to users exposed to the A/B test. It turned out that a bad config parameter value used for one of the A/B test groups caused `IndexOutOfBoundsException`. The spike and recovery of the app crash is shown in Figure 6.

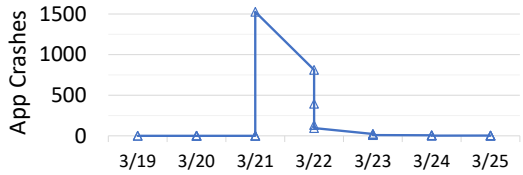

Figure 6: A canary test caught app crashes.

In the second example, a developer set up an A/B test to enable a new code path in an Android app, which increased memory consumption. Because the increase in memory consumption was quite moderate, it caused out-of-memory crashes on only a small fraction of very low-end devices with limited memory. The second-stage canary caught this subtle problem and prevented a silent regression in memory consumption.

7.6 Client-server Protocol Optimization

§6.3 describes MobileConfig's various network optimizations: query hashing (QH), value hashing (VH), and boolean encoding (BE). To evaluate their effectiveness, we measured the request and response size when fetching all configs for several apps under different settings. The results are summarized in Table 11, where *"All ON"* means enabling all optimizations, which is the setting used in production. Other settings mean disabling certain optimizations from the production baseline. These experiments are performed in a local testbed because disabling optimizations in production would cause a poor experience to real users.

Compared with *"All OFF"*, *"All ON"* reduces the total size of request and response by ≈96%. Value hashing and boolean encoding reduce the response size by about two orders of magnitude, while query hashing reduces the request size by ≈80%. The effectiveness of these optimizations is an important reason why we prefer the simple soft-state client-server protocol over a more complex hard-state protocol.

8 Operational Experiences

We use several production incidents to highlight the challenges and then share the lessons learned.

8.1 Production Incidents

Incident 1: lack of emergency push (EP). In 2016, a developer mistakenly changed the config parameter that controls the maximum number of comments to fetch in the *MM* app from 25 to 0, causing users to see no comments. The parameter value was quickly fixed on the server side, but as EP was not implemented in MobileConfig at the time, we had to painfully wait for over a day for most devices to gradually restart the app and apply the corrected parameter value. This incident expedited our development of EP.

Incident 2: EP malfunction. Early on, we observed that some devices had corrupted storage, leading to incorrect appearances of many config files. Initially, each use of EP fixed a single broken config parameter, and it was never the case that many configs needed simultaneous correction through EP. Given this, we adjusted the MobileConfig runtime to treat the situation of too many EP-delivered configs on storage as a sign of storage corruption, causing it to revert to default values for all configs. This precaution served us well for years until we began conducting large-scale Defcon drills [25]. These drills involved using EP to instruct apps to rapidly disable less-essential features to reduce the load on backend systems. In 2021, as Defcon drills expanded to disable more app features simultaneously, the number of configs pushed out by EP exceeded a threshold, triggering the MobileConfig runtime to mistakenly conclude that the storage was corrupted on many devices and fall back to default values for all configs. This incident demonstrates that as the operating environment shifts over time, an initial defensive mechanism can unexpectedly transform into a destructive force. Therefore, a robust solution needs to stand the test of time.

Incident 3: mishandling of file names. We once noticed network connection anomalies from about 0.1% of users due

to MobileConfig randomly flipping some config parameters. As the problem was not reproducible in our lab, initially, we made no progress after 27 days of laborious investigation. Eventually, a breakthrough occurred when the problem manifested on an employee's device, allowing us to directly attach a debugger. It turned out that when the app starts, if the latest config file's sequence number happens to end with 0, such as `10.ctable` or `20.ctable`, the MobileConfig runtime code, *"if (latestConfigFile.endsWith("0.ctable")),"* mistakenly treated it as `0.ctable`. The fix was simple—just changing the code to *"if (latestConfigFile.endsWith("/0.ctable"))"*—yet the investigation process was extremely difficult. This incident underscores the difficulty of developing low-level mobile systems like MobileConfig, exacerbated by the lack of direct access to user devices.

Incident 4: configs on VR devices. On our VR products, since we own the operating system, we run the MobileConfig runtime in a daemon to manage configs for all apps, which is more efficient than each app managing its own configs. When an app starts, it subscribes to the daemon for certain configs. While the solution overall worked well, it was reported that the daemon occasionally returned incorrect parameter values. As the problem was not reproducible in our lab, much of the investigation involved reading source code, changing code through trial and error, and waiting for logging data from user devices for confirmation. The whole process lasted 60 days. Eventually, it was discovered that when two apps concurrently subscribe to two sets of overlapping configs, the ordering of configs on storage depends on the timing of the subscription calls. This ordering issue was overlooked in some cases, resulting in retrieving wrong parameter values. This incident, once again, underscores the difficulties of developing low-level mobile systems.

8.2 Lessons Learned

We draw several lessons from our experience above.

- Following from Incident 1, we recommend every config framework to support emergency push. Although it is not supported by existing solutions in the public domain [1, 16, 28, 42] and the dozen legacy config frameworks predating MobileConfig at Meta, we found it important for enabling a safe and agile development process.
- Incidents 2, 3, and 4 all demonstrate the difficulty of developing a robust and feature-rich mobile config framework. Therefore, it should be done only once and then reused across all platforms, programming languages, and config use cases. This principle drives our design of the single, universal SDK, as opposed to Firebase's approach of using different implementations for different platforms.
- Incidents 3 and 4 also demonstrate that debugging subtle issues in the wild for low-level mobile systems like MobileConfig is very difficult due to a lack of access to user devices. Over time, we have enhanced consistency checking to detect various issues early (§5) and also upload snapshots of devices' config files to help us more easily reproduce problems.
- Finally, as shown in §7.5, traditional small-scale canary tests fall short for mobile apps due to noisy mobile environments. Consequently, MobileConfig employs unusually large and unusually long canary tests.

9 Related Work

Configuration management for mobile apps. Out of the few existing mobile config systems [1, 16, 28, 42], Google's Firebase RemoteConfig [16] is the closest to MobileConfig. A detailed comparison is shown in Table 8.

Configuration management for datacenter applications. Past studies on configuration management [10, 33–36, 38, 45, 50] have been mostly focused on datacenter applications. Configerator [38] is a representative system in the industry and a comparison is shown in Table 9.

Configuration error prevention. A large body of work studies misconfiguration [6, 9, 14, 24, 29, 43, 44, 46, 47, 49]. Like Configerator [38], MobileConfig primarily uses large-scale canary tests in production to prevent misconfiguration due to its robustness in complex environments, but MobileConfig has to address additional challenges in mobile environments such as ineffectiveness of small-scale canaries.

Consistency models. Out of many consistency models [2, 3, 23, 26, 31, 48], TACT [48] is most related to MobileConfig's progressive consistency. Both MobileConfig and TACT allow consumption of stale data and can bound the level of staleness. Config consumption on mobile devices can be viewed as read-only transactions, but neither related database work [8, 13] nor config consumption in datacenters [38] is optimized for accessing up-to-date data without blocking on remote reads.

10 Conclusion

We presented MobileConfig, a configuration management framework that manages tens of mobile apps on billions of devices. Its progressive consistency balances fast app startup with fresh config data. To prevent config errors, it uses a defense-in-depth approach that employs multi-stage canary tests at scale, compile-time validation, and config consistency checking. Its novel use of cross-platform code and server-side config parameter remapping prevents the proliferation of mobile SDKs while supporting diverse platforms, programming languages, and config use cases. Additionally, we reported our lessons learned from operating MobileConfig at hyperscale. Finally, we hope that our experience—for example, our developers making thousands of config changes daily in live production—will inspire others to also employ remote configs to push the limits of agile app development.

References

[1] Adobe Target: A/B Test, Personalize & Automate, 2024. https://business.adobe.com/products/target/adobe-target.html.

[2] Sarita V Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[4] App startup time, 2024. https://developer.android.com/topic/performance/vitals/launch-time.

[5] AppDynamics. Mobile app performance explained, 2014. https://www.appdynamics.com/media/uploaded-files/mobileapp.pdf.

[6] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the Ninth USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[8] Arvola Chan and Robert Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, (2):205–212, 1985.

[9] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. Test-case prioritization for configuration testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 452–465, 2021.

[10] Azure App Config, 2024. https://docs.microsoft.com/en-us/azure/azure-app-configuration/overview.

[11] Colin Contreary. Why should you care about your mobile app's startup time?, 2023. https://blog.embrace.io/why-should-you-care-about-your-mobile-apps-startup-time/.

[12] Feature toggle, 2024. https://en.wikipedia.org/wiki/Feature_toggle.

[13] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems (TODS)*, 7(2):209–234, 1982.

[14] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. ConfValley: A systematic configuration validation framework for cloud services. In *Proceedings of the 10th European Conference on Computer Systems*, page 19, 2015.

[15] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[16] Google Inc. Firebase Remote Config, 2024. https://firebase.google.com/docs/remote-config.

[17] Google Inc. Firebase Remote Config Loading Strategies, 2024. https://firebase.google.com/docs/remote-config/loading.

[18] Google Inc. Flatbuffers, 2024. https://google.github.io/flatbuffers/.

[19] Tyler Kieft. Building a better Instagram app for Android, 2014. https://instagram-engineering.com/building-a-better-instagram-app-for-android-c08f973662b.

[20] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1168–1176, 2013.

[21] Ron Kohavi, Diane Tang, and Ya Xu. *Trustworthy online controlled experiments: A practical guide to A/B testing*. Cambridge University Press, 2020.

[22] Ronny Kohavi, Thomas Crook, Roger Longbotham, Brian Frasca, Randy Henne, Juan Lavista Ferres, and Tamir Melamed. Online experimentation at Microsoft. *Data Mining Case Studies*, 11(2009):39, 2009.

[23] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[24] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–16, 2002.

[25] Justin J Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, et al. Defcon: Preventing Overload with Graceful Feature Degradation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, pages 607–622, 2023.

[26] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[27] Mike Nakhimovich. Improving Startup Time in the NYTimes Android App, 2016. https://archive.nytimes.com/open.blogs.nytimes.com/2016/02/11/improving-startup-time-in-the-nytimes-android-app/.

[28] Optimizely, 2024. https://www.optimizely.com/.

[29] Vasileios Pappas, Zhiguo Xu, Songwu Lu, Daniel Massey, Andreas Terzis, and Lixia Zhang. Impact of configuration errors on DNS robustness. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 319–330, 2004.

[30] Anshu Rustagi. How We Improved Our Android App "Cold Start" Time by 28%, 2018. https://redfin.engineering/how-we-improved-our-android-app-cold-start-time-by-28-a722e231314a.

[31] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[32] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[33] Gerald Schermann, Jürgen Cito, and Philipp Leitner. Continuous experimentation: challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31, 2018.

[34] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C Gall. Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Proceedings of the 17th International Middleware Conference*, pages 1–14, 2016.

[35] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430, 2016.

[36] Alex Sherman, Philip A Lisiecki, Andy Berkheimer, and Joel Wein. ACMS: The Akamai Configuration Management System. In *Proceedings of the Second USENIX Symposium on Networked Systems Design and Implementation*, pages 245–258, 2005.

[37] Snap Inc. Measuring 'Time to Camera ready', 2021. https://eng.snap.com/time_to_camera_ready.

[38] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[39] Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. Overlapping experiment infrastructure: More, better, faster experimentation. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 17–26, 2010.

[40] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994.

[41] Natansh Verma. Optimizing Facebook for iOS start time, 2015. https://engineering.fb.com/2015/11/20/ios/optimizing-facebook-for-ios-start-time/.

[42] VWO, 2024. https://vwo.com/.

[43] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pages 245–257, 2004.

[44] Avishai Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[45] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 619–634, 2016.

[46] Tianyin Xu and Yuanyuan Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys*, 47(4):70, 2015.

[47] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 159–172, 2011.

[48] Haifeng Yu. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[49] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th Architectural Support for Programming Languages and Operating Systems*, pages 687–700, 2014.

[50] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.