Holistic Configuration Management at Facebook

Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander,
Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl

Facebook Inc.

{tang, thawan, pradvenkat, akshay, wenzhe, aravindn, pdowell, robertkarl}@fb.com

Abstract

Facebook's web site and mobile apps are very dynamic. Every day, they undergo thousands of online configuration changes, and execute trillions of configuration checks to personalize the product features experienced by hundreds of million of daily active users. For example, configuration changes help manage the rollouts of new product features, perform A/B testing experiments on mobile devices to identify the best echo-canceling parameters for VoIP, rebalance the load across global regions, and deploy the latest machine learning models to improve News Feed ranking. This paper gives a comprehensive description of the use cases, design, implementation, and usage statistics of a suite of tools that manage Facebook's configuration end-to-end, including the frontend products, backend systems, and mobile apps.

1. Introduction

The software development and deployment cycle has accelerated dramatically [13]. A main driving force comes from the Internet services, where frequent software upgrades are not only possible with their service models, but also a necessity for survival in a rapidly changing and competitive environment. Take Facebook, for example. We roll *facebook.com* onto new code twice a day [29]. The site's various configurations are changed even more frequently, currently thousands of times a day. In 2014, thousands of engineers made live configuration updates to various parts of the site, which is even more than the number of engineers who made changes to Facebook's frontend product code.

Frequent configuration changes executed by a large population of engineers, along with unavoidable human mistakes, lead to configuration errors, which is a major source of site outages [24]. Preventing configuration errors is only one of the many challenges. This paper presents Facebook's holistic configuration management solution. Facebook uses Chef [7] to manage OS settings and software deployment [11], which is not the focus of this paper. Instead, we focus on the homegrown tools for managing applications' dynamic runtime configurations that may be updated live multiple times a day, without application redeployment or restart. Examples include gating product rollouts, managing application-level traffic, and running A/B testing experiments.

Below, we outline the key challenges in configuration management for an Internet service and our solutions.

Configuration sprawl. Facebook internally has a large number of systems, including frontend products, backend services, mobile apps, data stores, etc. They impose different requirements on configuration management. Historically, each system could use its own configuration store and distribution mechanism, which makes the site as a whole hard to manage. To curb the configuration sprawl, we use a suite of tools built on top of a uniform foundation to support the diverse use cases. Currently, the tools manage hundreds of thousands of *configs* (i.e., configuration files) from a central location, and distribute them to hundreds of thousands of servers and more than one billion mobile devices.

Configuration authoring and version control. A large-scale distributed system often has many flexible knobs that can be tuned live. The median size of a config at Facebook is 1KB, with large ones reaching MBs or GBs. Manually editing these configs is error prone. Even a minor mistake could potentially cause a site-wide outage. We take a truly *configuration-as-code* approach to compile and generate configs from high-level source code. We store the config programs and the generated configs in a version control tool.

Defending against configuration errors. We safeguard configs in multiple ways. First, the configuration compiler automatically runs *validators* to verify invariants defined for configs. Second, a config change is treated the same as a code change and goes though the same rigorous code review process. Third, a config change that affects the frontend products automatically goes through continuous integration tests in a sandbox. Lastly, the automated canary testing tool rolls out a config change to production in a staged fashion,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815401

monitors the system health, and rolls back automatically in case of problems. A main hurdle we have to overcome is to reliably determine the health of numerous backend systems.

Configuration dependency. Facebook.com is powered by a large number of systems developed by many different teams. Each system has its own config but there are dependencies among the configs. For example, after the monitoring tool's config is updated to enable a new monitoring feature, the monitoring configs of all other systems might need be updated accordingly. Our framework expresses configuration dependency as source code dependency, similar to the *include* statement in a *C++* program. The tool automatically extracts dependencies from source code without the need to manually edit a *makefile*.

Scalable and reliable configuration distribution. Our tools manage a site that is much larger than the previously reported configuration distribution system [30], and support a much more diverse set of applications, including mobile. The size of a config can be as small as a few bytes or as large as GBs. Given the scale and the geographically distributed locations, failures are the norm. It is a significant challenge to distribute configs to all servers and mobile devices in a timely and reliable manner, and not to make the availability of the configuration management tools become a bottleneck of the applications' availability.

In this paper, we describe our solutions to these challenges. We make the following contributions:

- Runtime configuration management is an important problem for Internet services, but it is not well defined in the literature. We describe the problem space and the real use cases from our experience, in the hope of motivating future research in this area.
- We describe Facebook's configuration management stack, which addresses many challenges not covered by prior work, e.g., gating product rollouts, config authoring, automated canary testing, mobile config, and a hybrid subscription-P2P model for large config distribution. This is the first published solution of holistic configuration management for Internet services.
- We report the statistics and experience of operating a large-scale configuration management stack, which are made available in the literature for the first time. For example, do old configs become dormant, and how often do config changes expose code bugs?

2. Use Cases and Solution Overview

We focus on the problem of managing an Internet service's dynamic runtime configurations that may be updated live multiple times a day, without application redeployment or restart. Configuration management is an overloaded term. We describe several real use cases to make the problem space more concrete. Note that they are just a tiny sample set out of the hundreds of thousands of configs we manage today.

Gating new product features. Facebook releases software early and frequently. It forces us to get early feedback and iterate rapidly. While a new product feature is still under development, we commonly release the new code into production early but in a disabled mode, and then use a tool called Gatekeeper to incrementally enable it online. Gatekeeper can quickly disable the new code if problems arise. It controls which users will experience the new feature, e.g., Facebook employees only or 1% of the users of a mobile device model. The target can be changed live through a config update.

Conducting experiments. Good designs often require A/B tests to guide data-driven decision making. For example, the echo-canceling parameters for VoIP on Facebook Messenger need tuning for different mobile devices because of the hardware variation. Our tools can run live experiments in production to test different parameters through config changes.

Application-level traffic control. Configs help manage the site's traffic in many ways. Automation tools periodically make config changes to shift traffic across regions and perform load tests in production. In case of emergency, a config change kicks off automated cluster/region traffic drain and another config change disables resource-hungry features of the site. During shadow tests, a config change starts or stops duplicating live traffic to testing servers. During a fire drill, a config change triggers fault injection into a production system to evaluate its resilience.

Topology setup and load balancing. Facebook stores user data in a large-scale distributed data store called TAO [5]. As the hardware setup changes (e.g., a new cluster is brought online), the macro traffic pattern shifts, or failure happens, the application-level configs are updated to drive topology changes for TAO and rebalance the load.

Monitoring, alerts, and remediation. Facebook's monitoring stack is controlled through config changes: 1) what monitoring data to collect, 2) monitoring dashboard (e.g., the layout of the key-metric graphs), 3) alert detection rules (i.e., what is considered an anomaly), 4) alert subscription rules (i.e., who should be paged), and 5) automated remediation actions [27], e.g., rebooting or reimaging a server. All these can be dynamically changed without a code upgrade, e.g., as troubleshooting requires collecting more monitoring data.

Updating machine learning models. Machine learning models are used to guide search ranking, News Feed ranking, and spam detection. The models are frequently retrained with the latest data and distributed to the servers without a code upgrade. This kind of model is used for many products. Its data size can vary from KBs to GBs.

Controlling an application's internal behavior. This is one of the most common use cases. A production system often has many knobs to control its behavior. For example, a data store's config controls how much memory is reserved for caching, how many writes to batch before writing to the disk, how much data to prefetch on a read, etc.

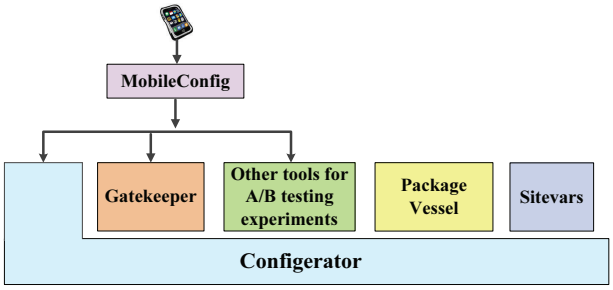

Figure 1: Facebook's configuration management tools. *MobileConfig* supports mobile apps. All the other tools support applications running in data centers.

2.1 Tool Overview

Figure 1 shows Facebook's configuration management tools. They work together to support the diverse use cases.

Configerator provides all the foundational functions, including version control, authoring, code review, automated canary testing, and config distribution. Other tools are built on top of Configerator and provide specialized functions.

Gatekeeper controls the rollouts of new product features. Moreover, it can also run A/B testing experiments to find the best config parameters. In addition to Gatekeeper, Facebook has other A/B testing tools built on top of Configerator, but we omit them in this paper due to the space limitation.

PackageVessel uses peer-to-peer file transfer to assist the distribution of large configs (e.g., GBs of machine learning models), without sacrificing the consistency guarantee.

Sitevars is a shim layer that provides an easy-to-use configuration API for the frontend PHP products.

MobileConfig manages mobile apps' configs on Android and iOS, and bridges them to the backend systems such as Configerator and Gatekeeper. MobileConfig is not bridged to Sitevars because Sitevars is for PHP only. MobileConfig is not bridged to PackageVessel because currently there is no need to transfer very large configs to mobile devices.

We describe these tools in the following sections.

3. Configerator, Sitevars, and PackageVessel

Among other things, Configerator addresses the challenges in configuration authoring, configuration error prevention, and large-scale configuration distribution.

3.1 Configuration Authoring

Our hypotheses are that 1) most engineers prefer writing code to generate a config (i.e., a configuration file) instead of manually editing the config, and 2) most config programs are easier to maintain than the raw configs themselves. We will use data to validate these hypotheses in Section 6.1.

Following these hypotheses, Configerator literally treats "*configuration as code*". Figure 2 shows an example. A config's data schema is defined in the platform-independent Thrift [2] language (see "*job.thrift*"). The engineer writes two Python files "*create_job.cinc*" and "*cache_job.cconf*" to manipulate the Thrift object. A call to "*export_if_last()*" writes the config as a JSON [20] file. To prevent invalid configs, the engineer writes another Python file "*job.thrift-cvalidator*" to express the invariants for the config. The validator is automatically invoked by the Configerator compiler to verify every config of type "*Job*".

The source code of config programs and generated JSON configs are stored in a version control tool, e.g., git [14]. In the upper-left side of Figure 3, the engineer works in a "*development server*" with a local clone of the git repository. She edits the source code and invokes the Configerator compiler to generate JSON configs. At the top of Figure 3, config changes can also be initiated by an engineer via a Web UI, or programmatically by an automation tool invoking the APIs provided by the "*Mutator*" component.

The example in Figure 2 separates *create_job.cinc* from *cache_job.cconf* so that the former can be reused as a common module to create configs for other types of jobs. Hypothetically, three different teams may be involved in writing the config code: the scheduler team, the cache team, and the security team. The scheduler team implements the scheduler software and provides the shared config code, including the config schema *job.thrift*, the reusable module *create_job.cinc*, and the validator *job.thrift-cvalidator*, which ensures that configs provided by other teams do not accidentally break the scheduler. The cache team generates the config for a cache job by simply invoking *create_job(name="cache")*, while the security team generates the config for a security job by simply invoking *create_job(name="security")*.

Code modularization and reuse are the key reasons why maintaining config code is easier than manually editing JSON configs. Config dependencies are exposed as code dependencies through *import_thrift()* and *import_python()*. An example is shown below.

Python file "app_port.cinc"

```
APP_PORT = 8089
```

Python file "app.cconf"

```
import_python ("app_port.cinc", "*")
app_cfg = AppConfig (port = APP_PORT …)
export_if_last (app_cfg)
```

Python file "firewall.cconf"

```
import_python ("app_port.cinc", "*")
firewall_cfg = FirewallConfig (port = APP_PORT …)
export_if_last (firewall_cfg)
```

"*app.cconf*" instructs an application to listen on a specific port. "*firewall.cconf*" instructs the OS to allow traffic on that port. Both depend on "*app_port.cinc*". The "*Dependency Service*" in Figure 3 automatically extracts dependencies from source code. If *APP_PORT* in "*app_port.cinc*" is changed, the Configerator compiler automatically recompiles both "*app.cconf*" and "*firewall.cconf*", and updates

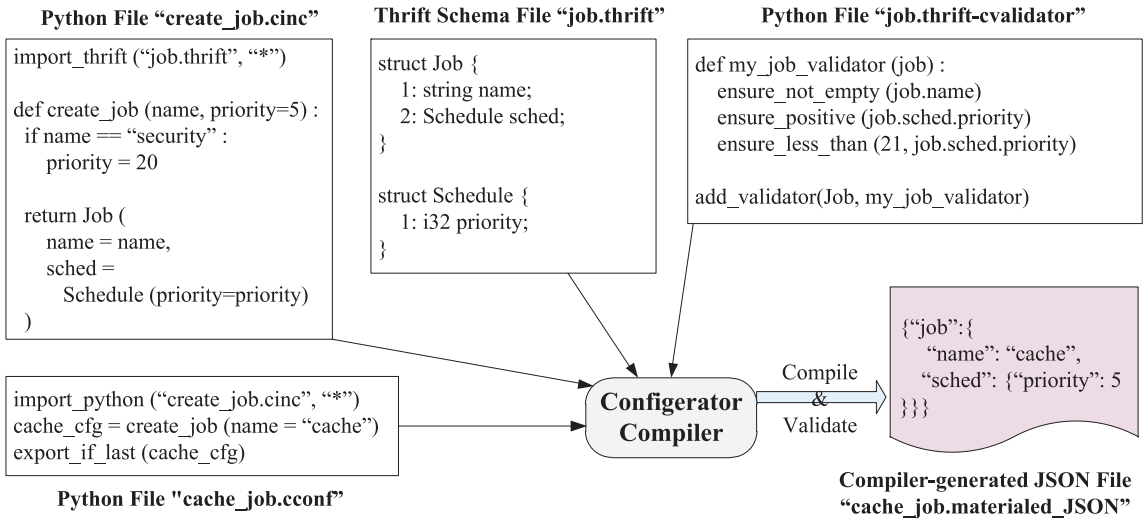

Figure 2: The Configerator compiler generates a JSON configuration file from the Python and Thrift source code.

their JSON configs in one git commit, which ensures consistency. Dependency can be expressed using any Python language construct, not limited to shared constants.

3.2 Improving Usability through UI and Sitevars

Configerator is designed as a uniform platform to support all use cases. It must be sufficiently flexible and expressive in order to support complex configs. On the other hand, simple configs may not benefit much from the complexity of the Python and Thrift code in Figure 2. The Configerator UI allows an engineer to directly edit the value of a Thrift config object without writing any code. The UI automatically generates the artifacts needed by Configerator.

The *Sitevars* tool is a shim layer on top of Configerator to support simple configs used by frontend PHP products. It provides configurable name-value pairs. The value is a PHP expression. An engineer uses the Sitevars UI to easily update a sitevar's PHP content without writing any Python/Thrift code. A sitevar can have a *checker* implemented in PHP to verify the invariants, similar to the validator in Figure 2. Because PHP is weakly typed, sitevars are more prone to configuration errors, e.g., typos. Engineers are encouraged to define a data schema for a newly created sitevar. A legacy sitevar may predate this best practice. The tool automatically infers its data type from its historical values. For example, it infers whether a sitevar's field is a string. If so, it further infers whether it is a JSON string, a timestamp string, or a general string. If a sitevar update deviates from the inferred data type, the UI displays a warning message to the engineer.

3.3 Preventing Configuration Errors

Configuration errors are a major source of site outages [24]. We take a holistic approach to prevent configuration errors, including 1) config validators to ensure that invariants are not violated, 2) code review for both config programs and generated JSON configs, 3) manual config testing, 4) automated integration tests, and 5) automated canary tests. They complement each other to catch different configuration errors. We follow the flow in Figure 3 to explain them.

To manually test a new config, an engineer runs a command to temporarily deploy the new config to some production servers or testing servers, and verifies that everything works properly. Once satisfied, the engineer submits the source code, the JSON configs, and the testing results to a code review system called Phabricator [26]. If the config is related to the frontend products of *facebook.com*, in a sandbox environment, the “*Sandcastle*” tool automatically performs a comprehensive set of synthetic, continuous integration tests of the site under the new config. Sandcastle posts the testing results to Phabricator for reviewers to access. Once the reviewers approve, the engineer pushes the config change to the remote “*Canary Service*”.

The canary service automatically tests a new config on a subset of *production machines* that serve live traffic. It complements manual testing and automated integration tests. Manual testing can execute tests that are hard to automate, but may miss config errors due to oversight or shortcut under time pressure. Continuous integration tests in a sandbox can have broad coverage, but may miss config errors due to the small-scale setup or other environment differences.

A config is associated with a canary spec that describes how to automate testing the config in production. The spec defines multiple testing phases. For example, in phase 1, test on 20 servers; in phase 2, test in a full cluster with thousands of servers. For each phase, it specifies the testing target servers, the healthcheck metrics, and the predicates that decide whether the test passes or fails. For example, the click-through rate (CTR) collected from the servers using the new config should not be more than $x\%$ lower than the CTR collected from the servers still using the old config.

The canary service talks to the “*Proxies*” running on the servers under test to temporarily deploy the new config (see

the bottom of Figure 3). If the new config passes all testing phases, the canary service asks the remote "*Landing Strip*" to commit the change into the master git repository.

3.4 Scalable and Reliable Configuration Distribution

Configerator distributes a config update to hundreds of thousands of servers scattered across multiple continents. In such an environment, failures are the norm. In addition to scalability and reliability, other properties important to Configerator are 1) availability (i.e., an application should continue to run regardless of failures in the configuration management tools); and 2) data consistency (i.e., an application's instances running on different servers should eventually receive all config updates delivered in the same order, although there is no guarantee that they all receive a config update exactly at the same time). In this section, we describe how Configerator achieves these goals through the push model.

In Figure 3, the git repository serves as the ground truth for committed configs. The "*Git Tailer*" continuously extracts config changes from the git repository, and writes them to *Zeus* for distribution. Zeus is a forked version of ZooKeeper [18], with many scalability and performance enhancements in order to work at the Facebook scale. It runs a consensus protocol among servers distributed across multiple regions for resilience. If the leader fails, a follower is converted into a new leader.

Zeus uses a three-level high-fanout distribution tree, *leader→observer→proxy*, to distribute configs through the push model. The leader has hundreds of observers as children in the tree. A high-fanout tree is feasible because the data-center network has high bandwidth and only small data is sent through the tree. Large data is distributed through a peer-to-peer protocol separately (see Section 3.5). The three-level tree is simple to manage and sufficient for the current scale. More levels can be added in the future as needed.

Each Facebook data center consists of multiple clusters. Each cluster consists of thousands of servers, and has multiple servers designated as Zeus *observers*. Each observer keeps a fully replicated read-only copy of the leader's data. Upon receiving a write, the leader commits the write on the followers, and then asynchronously pushes the write to each observer. If an observer fails and then reconnects to the leader, it sends the latest transaction ID it is aware of, and requests the missing writes. The commit log of ZooKeeper's consensus protocol helps guarantee in-order delivery of config changes.

Each server runs a Configerator "*Proxy*" process, which randomly picks an observer in the same cluster to connect to. If the observer fails, the proxy connects to another observer. Unlike an observer, the proxy does not keep a full replica of the leader's data. It only fetches and caches the configs needed by the applications running on the server.

An application links in the Configerator client library to access its config. On startup, the application requests the proxy to fetch its config. The proxy reads the config from

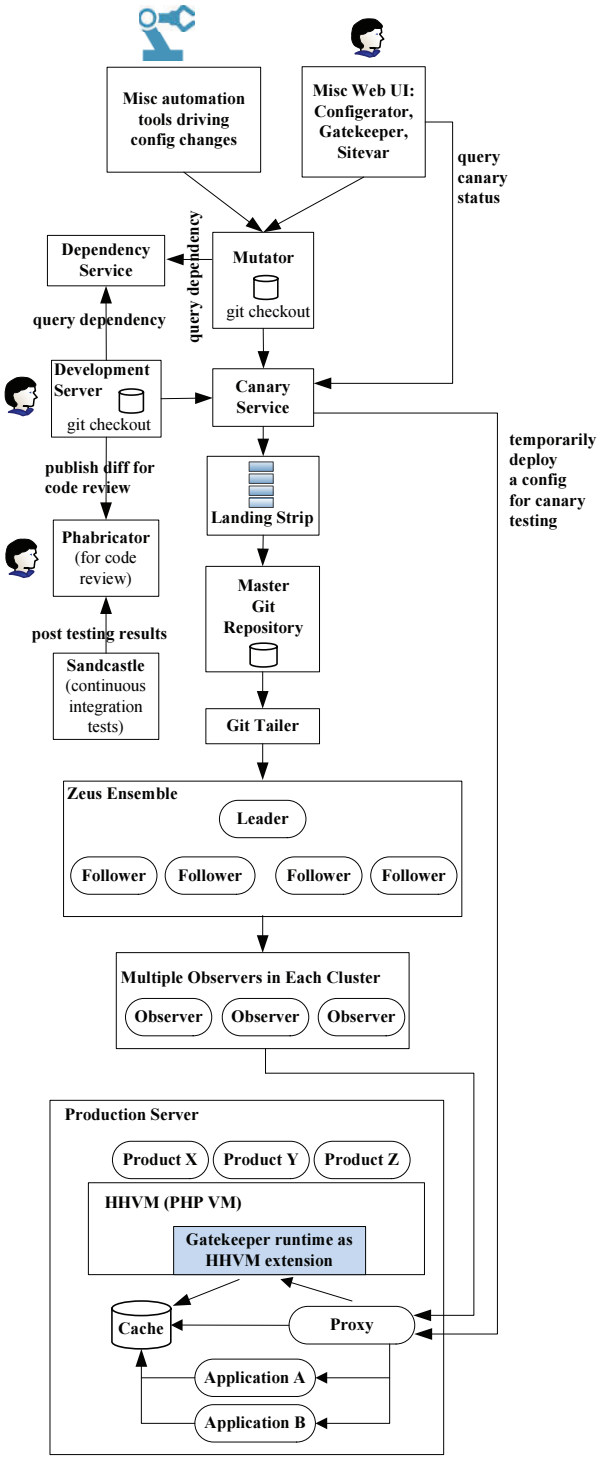

Figure 3: **Architecture of Configerator**. It uses *git* for version control, *Zeus* for tree-structured config distribution, *Phabricator* for code review, *Sandcastle* for continuous integration tests, *Canary Service* for automated canary tests, *Landing Strip* for committing changes, *Mutator* for providing APIs to support automation tools, and *Dependency Service* for tracking config dependencies. Applications on a production server interact with the *Proxy* to access their configs.

the observer with a watch so that later the observer will notify the proxy if the config is updated. The proxy stores the config in an on-disk cache for later reuse. If the proxy fails, the application falls back to read from the on-disk cache directly. This design provides high availability. So long as a config exists in the on-disk cache, the application can access it (though outdated), even if all Configerator components fail, including the git repository, Zeus leader/followers, observers, and proxy.

Configerator uses the push model. How does it compare with the pull model [19, 30]? The biggest advantage of the pull model is its simplicity in implementation, because the server side can be stateless, without storing any hard state about individual clients, e.g., the set of configs needed by each client (note that different machines may run different applications and hence need different configs). However, the pull model is less efficient for two reasons. First, some polls return no new data and hence are pure overhead. It is hard to determine the optimal poll frequency. Second, since the server side is stateless, the client has to include in each poll the full list of configs needed by the client, which is not scalable as the number of configs grows. In our environment, many servers need tens of thousands of configs to run. We opt for the push model in our environment.

3.5 Distributing Large Configs through PackageVessel

Some configs can be large, e.g., machine learning models for News Feed ranking. As these large configs may change frequently, it is not scalable to deliver them through Zeus' distribution tree, because it would overload the tree's internal nodes that have a high fan out. Moreover, it is hard to guarantee the quality of service if the distribution paths overlap between large configs and small (but critical) configs.

Our *PackageVessel* tool solves the problem by separating a large config's metadata from its bulk content. When a large config changes, its bulk content is uploaded to a storage system. It then updates the config's small metadata stored in Configerator, including the version number of the new config and where to fetch the config's bulk content. Configerator guarantees the reliable delivery of the metadata to servers that subscribe to the config. After receiving the metadata update, a server fetches the config's bulk content from the storage system using the BitTorrent [8] protocol. Servers that need the same large config exchange the config's bulk content among themselves in a peer-to-peer (P2P) fashion to avoid overloading the centralized storage system. Our P2P protocol is locality aware so that a server prefers exchanging data with other servers in the same cluster. We recommend using PackageVessel for configs larger than 1MB.

A naive use of P2P cannot guarantee data consistency. Our hybrid subscription-P2P model does not have this limitation. Zeus' subscription model guarantees the consistency of the metadata, which in turn drives the consistency of the bulk content. For example, Facebook's spam-fighting system updates and distributes hundreds of MBs of config data to thousands of global servers many times a day. Our statistics show that PackageVessel consistently and reliably delivers the large configs to the live servers in less than four minutes.

3.6 Improving Commit Throughput

Multiple engineers making concurrent config commits into a shared git repository causes contention and slows down the commit process. We explain it through an example. When an engineer tries to push a config diff X to the shared git repository, git checks whether the local clone of the repository is up to date. If not, she has to first bring her local clone up to date, which may take 10s of seconds to finish. After the update finishes, she tries to push diff X to the shard repository again, but another diff Y from another engineer might have just been checked in. Even if diff X and diff Y change different files, git considers the engineer's local repository clone outdated, and again requires an update.

The "*Landing Strip*" in Figure 3 alleviates the problem, by 1) receiving diffs from committers, 2) serializing them according to the first-come-first-served order, and 3) pushing them to the shared git repository on behalf of the committers, without requiring the committers to bring their local repository clones up to date. If there is a true conflict between a diff being pushed and some previously committed diffs, the shared git repository rejects the diff, and the error is relayed back to the committer. Only then, the committer has to update her local repository clone and resolve the conflict.

The landing strip alleviates the commit-contention problem, but does not fundamentally solve the commit-throughput problem, because 1) a shared git repository can only accept one commit at a time, and 2) git operations become slower as the repository grows larger. Configerator started with a single shared git repository. To improve the commit throughput, we are in the process of migration to multiple smaller git repositories that collectively serve a partitioned global name space. Files under different paths (e.g., "*/feed*" and "*/tao*") can be served by different git repositories that can accept commits concurrently. Cross-repository dependency is supported.

```
import_python(``feed/A.cinc'', ``*'')
import_python(``tao/B.cinc'', ``*'')
...
```

In the example above, the config imports two other configs. The code is the same regardless of whether those configs are in the same repository or not. The Configerator compiler uses metadata to map the dependent configs to their repositories and automatically fetches them if they are not checked out locally. Along with the partitioning of the git repositories, the components in Figure 3 are also partitioned. Each git repository has its own mutator, landing strip, and tailer.

New repositories can be added incrementally. As a repository grows large, some of its files can be migrated into a new repository. It only requires updating the metadata that

lists all the repositories and the file paths they are responsible for. The contents of the migrated files require no change.

3.7 Fault Tolerance

Every component in Figure 3 has built-in redundancy across multiple regions. One region serves as the master. Each backup region has its own copy of the git repository, and receives updates from the master region. The git repository in a region is stored on NFS and mounted on multiple servers, with one as the master. Each region runs multiple instances of all the services, including mutator, canary service, landing strip, and dependency service. Configerator supports failover both within a region and across regions.

3.8 Summary

Configerator addresses the key challenges in configuration authoring, configuration error prevention, and configuration distribution. It takes a configuration-as-code approach to compile and generate configs from high-level source code. It expresses configuration dependency as source code dependency, and encourages config code modularization and reuse. It takes a holistic approach to prevent configuration errors, including config validators, code review, manual config testing, automated integration tests, and automated canary tests. It uses a distribution tree to deliver small configs through the push model, and uses a P2P protocol to deliver the bulk contents of large configs. It avoids commit contention by delegating commits to the landing strip. It improves commit throughput by using multiple git repositories that collectively serve a partitioned global name space.

4. Gatekeeper

Facebook releases software early and frequently. It forces us to get early feedback and iterate rapidly. It makes troubleshooting easier, because the delta between two releases is small. It minimizes the use of code branches that complicate maintenance. On the other hand, frequent software releases increase the risk of software bugs breaking the site. This section describes how Gatekeeper helps mitigate the risk by managing code rollouts through online config changes.

While a new product feature is still under development, Facebook engineers commonly release the new code into production early but in a disabled mode, and then use Gatekeeper to incrementally enable it online. If any problem is detected during the rollout, the new code can be disabled instantaneously. Without changing any source code, a typical launch using Gatekeeper goes through multiple phases. For example, initially Gatekeeper may only enable the product feature to the engineers developing the feature. Then Gatekeeper can enable the feature for an increasing percentage of Facebook employees, e.g., 1%→10%→100%. After successful internal testing, it can target 5% of the users from a specific region. Finally, the feature can be launched globally with an increasing coverage, e.g., 1%→ 10%→100%.

```
if(gk_check(``ProjectX'', $user_id)) {
  // Show the new feature to the user.
  ...
} else {
  // Show the old product behavior.
  ...
}
```

Figure 4: Pseudocode of a product using Gatekeeper to control the rollout of a product feature.

```
bool gk_check($project, $user_id) {
  if ($project == ``ProjectX'') {
    // The gating logic for ``ProjectX''.
    if($restraint_1($user_id) AND
       $restraint_2($user_id)) {
      //Cast the die to decide pass or fail.
      return rand($user_id) < $pass_prob_1;
    } else if ($restraint_4($user_id)) {
      return rand($user_id) < $pass_prob_2;
    } else {
      return false;
    }
  ...
}
```

Figure 5: Pseudocode of Gatekeeper's gating logic.

Figure 4 shows how a piece of product code uses a Gatekeeper "*project*" (i.e., a specific gating logic) to enable or disable a product feature. Figure 5 shows the project's internal logic, which consists of a series of *if-then-else* statements. The condition in an *if*-statement is a conjunction of predicates called *restraints*. Examples of restraints include checking whether the user is a Facebook employee and checking the type of a mobile device. Once an *if*-statement is satisfied, it probabilistically determines whether to pass or fail the gate, depending on a configurable probability that controls user sampling, e.g., 1% or 10%.

The code in Figure 5 is conceptual. A Gatekeeper project's control logic is actually stored as a config that can be changed live without a code upgrade. Through a Web UI, the *if-then-else* statements can be added or removed (actually, it is a graphical representation, without code to write[1]); the restraints in an *if*-statement can be added or removed; the probability threshold can be modified; and the parameters specific to a restraint can be updated. For example, the user IDs in the "*ID()*" restraint can be added or removed so that only specific engineers will experience the new product feature during the early development phase.

A Gatekeeper project is dynamically composed out of restraints through configuration. Internally, a restraint is statically implemented in PHP or C++. Currently, hundreds of

[1] Code review is supported even if changes are made through UI. The UI tool converts a user's operations on the UI into a text file, e.g., "*Updated Employee sampling from 1% to 10%*". The text file and a screenshot of the config's final graphical representation are submitted for code review.

restraints have been implemented, which are used to compose tens of thousands of Gatekeeper projects. The restraints check various conditions of a user, e.g., country/region, locale, mobile app, device, new user, and number of friends.

A Gatekeeper project's control logic is stored as a JSON config in Configerator. When the config is changed (e.g., expanding the rollout from 1% to 10%), the new config is delivered to production servers (see the bottom of Figure 3). The Gatekeeper runtime reads the config and builds a boolean tree to represent the gating logic. Similar to how an SQL engine performs cost-based optimization, the Gatekeeper runtime can leverage execution statistics (e.g., the execution time of a restraint and its probability of returning true) to guide efficient evaluation of the boolean tree.

The example in Figure 5 is similar to the disjunctive normal form (DNF), except the use of *rand()* to sample a subset of users. Sampling is inherent to feature gating, i.e., rolling out a feature to an increasingly larger population, e.g., $1\% \rightarrow 10\%$. The negation operator is built inside each restraint. For example, the employee restraint can be configured to check "*not an employee*". As a result, the gating logic has the full expressive power of DNF.

Gatekeeper uses DNF of restraints and user sampling to form the gating logic. It strikes a balance among flexibility, usability, and safety. Theoretically, it is possible not to impose any structure on the gating logic (i.e., not limited to the form in Figure 5), by allowing an engineer to write arbitrary gating code in a dynamic programming language (e.g., PHP), and immediately distributing it for execution as a live config update. This approach offers maximum flexibility, but increases the risk of configuration errors. Moreover, it is harder to use for most engineers, compared with simply selecting restraints from Gatekeeper's UI without writing any code. Finally, its additional flexibility over Gatekeeper is limited, because Facebook rolls out PHP code twice a day and new restraints can be added quickly.

Some gating logic is computationally too expensive to execute realtime inside a restraint. In one example, a product feature should only be exposed to users whose recent posts are related to the current trending topics. This computation requires continuous stream processing. In another example, it needs to run a MapReduce job to analyze historical data to identify users suitable for a product feature. Gatekeeper provides a key-value-store interface to integrate with these external systems. A special "*laser()*" restraint invokes *get("$project-$user_id")* on a key-value store called *Laser*. If the return value is greater than a configurable threshold, i.e., *get(...)>T*, the restraint passes. Any system can integrate with Gatekeeper by putting data into Laser. Laser stores data on flash or in memory for fast access. It has automated data pipelines to load data from the output of a stream processing system or a MapReduce job. The MapReduce job can be re-run periodically to refresh the data for all users.

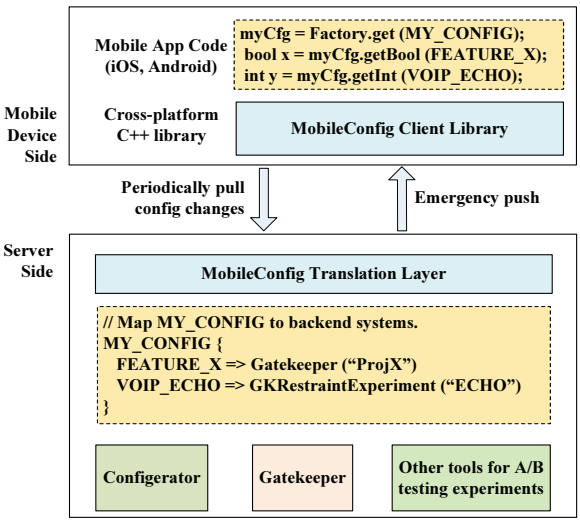

Figure 6: MobileConfig architecture.

5. MobileConfig

Configuration management for mobile apps differs from that for applications running in data centers, because of the unique challenges in a mobile environment. First, the mobile network is a severe limiting factor. Second, mobile platforms are diverse, with at least Android and iOS to support. Lastly, legacy versions of a mobile app linger around for a long time, raising challenges in backward compatibility.

MobileConfig addresses these challenges while maximizing the reuse of the configuration management tools already developed for applications running in data centers. Figure 6 shows the architecture of MobileConfig. Every config appears as a *context class* in a mobile app's native language. The app invokes the context class' getter methods to retrieve the values of the config fields. The client library that supports the context class is implemented in C++ so that it is portable across Android and iOS.

Because push notification is unreliable, MobileConfig cannot solely rely on the push model for config distribution. The client library polls the server for config changes (e.g., once every hour) and caches the configs on flash for later reuse. To minimize the bandwidth consumption, the client sends to the server the hash of the config schema (for schema versioning) and the hash of the config values cached on the client.[2] The server sends back only the configs that have changed and are relevant to the client's schema version. In addition to pull, the server occasionally pushes emergency config changes to the client through push notification, e.g., to immediately disable a buggy product feature. A combination of push and pull makes the solution simple and reliable.

To cope with legacy versions of a mobile app, separating abstraction from implementation is a first-class citizen

[2] One future enhancement is to make the server stateful, i.e., remembering each client's hash values to avoid repeated transfer of the hash values.

in MobileConfig. The *translation layer* in Figure 6 provides one level of indirection to flexibly map a MobileConfig field to a backend config. The mapping can change. For example, initially *VOIP_ECHO* is mapped to a Gatekeeper-backed experiment, where satisfying different *if*-statements in Figure 5 gives *VOIP_ECHO* a different parameter value to experiment with. After the experiment finishes and the best parameter is found, *VOIP_ECHO* can be remapped to a constant stored in Configerator. In the long run, all the backend systems (e.g., Gatekeeper and Configerator) may be replaced by new systems. It only requires changing the mapping in the translation layer to smoothly finish the migration. To scale to more than one billion mobile devices, the translation layer runs on many servers. The translation mapping is stored in Configerator and distributed to all the translation servers.

6. Usage Statistics and Experience

We described Facebook's configuration management stack. Given the space limit, this section will primarily focus on the usage statistics of the production systems and our experience, because we believe those production data are more valuable than experimental results in a sandbox environment. For the latter, we report Configerator's commit-throughput scalability test in a sandbox, because the data cannot be obtained from production, and currently commit throughput is Configerator's biggest scalability challenge.

We attempt to answer the following questions:

- Does the *configuration-as-code* hypothesis hold, i.e., most engineers prefer writing config-generating code?
- What are the interesting statistics about config update patterns, e.g., do old configs become dormant quickly?
- What are the performance and scale of the tools?
- What are the typical configuration errors?
- How do we scale our operation to support thousands of engineers and manage configs on hundreds of thousands of servers and a billion or more mobile devices?
- How does an organization's engineering culture impact its configuration management solution?

6.1 Validating the Configuration-as-Code Hypothesis

Configerator stores different types of files: 1) the Python and Thrift source code as shown in Figure 2; 2) the "*compiled configs*", i.e., the JSON files generated by the Configerator compiler from the source code; 3) the "*raw configs*", which are not shown in Figure 2 for brevity. Configerator allows engineers to check in raw configs of any format. They are not produced by the Configerator compiler, but are distributed in the same way as compiled configs. They are either manually edited or produced by other automation tools.

Figure 7 shows the number of configs stored in Configerator. The growth is rapid. Currently it stores hundreds of thousands of configs. The compiled configs grow faster than the raw configs. Out of all the configs, 75% of them are

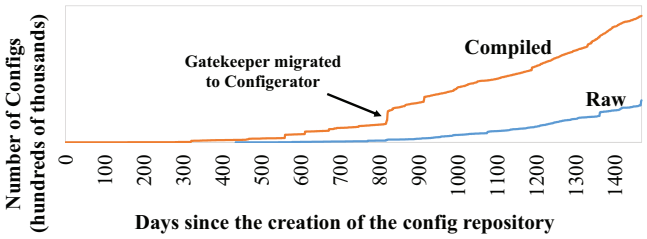

Figure 7: Number of configs in the repository. The scale on the y-axis is removed due to confidentiality, but we keep the unit to show the order of magnitude.

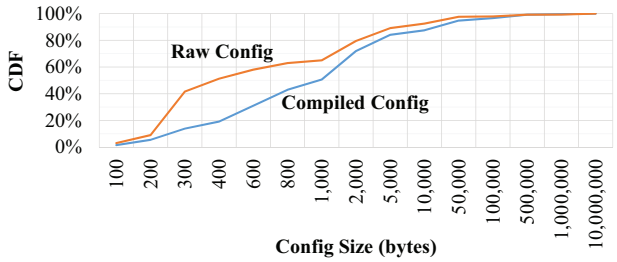

Figure 8: Cumulative distribution function (CDF) of config size. Note that the x-axis is neither linear nor logscale.

compiled configs. Moreover, about 89% of the updates to raw configs are done by automation tools, i.e., not manually edited. *This validates one important hypothesis of Configerator, i.e., engineers prefer writing config-generating code to manually editing configs.* The custom automation tools are used to generate raw configs because they suit the specific (often simpler) jobs better, or they predate Configerator.

Figure 8 shows the CDF of config size. Many configs have significant complexity and are not trivial name-value pairs. *The complexity is one reason why engineers prefer not editing configs manually.* Compiled configs are more complex than raw configs. The P50's of raw config size and compiled config size are 400 bytes and 1KB, respectively. The P95's are 25KB and 45KB, respectively. The largest configs are 8.4MB and 14.8MB, respectively. Even larger configs are distributed through PackageVessel and only their small metadata is stored in Configerator.

On average, a raw/compiled/source config gets updated 44/16/10 times during its lifetime, respectively. Raw configs get updated 175% more frequently than compiled configs, because most (89%) of the raw config changes are done by automation tools. Compiled configs are generated from config source code, but the former changes 60% more frequently than the latter does, because the change of one source code file may generate multiple new compiled configs, similar to a header file change in C++ causing recompilation of multiple .cpp files. *This indicates that writing code to generate configs reduces the burden of manually keeping track of changes, thanks to code reuse.*

6.2 Config Update Statistics

This section reports config update statistics, which hopefully can help guide the design of future configuration management systems, and motivate future research.

Are configs fresh or dormant, i.e., are they updated recently? Figure 9 shows that both fresh configs and dormant configs account for a significant fraction. Specifically, 28% of the configs are either created or updated in the past 90 days. On the other hand, 35% of the configs are not updated even once in the past 300 days.

Do configs created a long time ago still get updated? Figure 10 shows that both new configs and old configs get updated. 29% of the updates happen on configs created in the past 60 days. On the other hand, 29% of the updates happen on configs created more than 300 days ago, which is significant because those old configs only account for 50% of the configs currently in the repository. The configs do not stabilize as quickly as we initially thought.

The frequency of config updates is highly skewed, as shown in Table 1. It is especially skewed for raw configs because their commits are mostly done by automation tools. At the low end, 56.9% of the raw configs are created but then never updated. At the high end, the top 1% of the raw configs account for 92.8% of the total updates to raw configs. By contrast, 25.0% of the compiled configs are created but then never updated, and the top 1% of the compiled configs account for 64.5% of the total updates.

When a config gets updated, is it a big change or a small change? Table 2 shows that most changes are very small. In the output of the Unix *diff* tool, it is considered a one-line change to add a new line or delete an existing line in a file. Modifying an existing line is considered a two-line change: first deleting the existing line, and then adding a new line. Table 2 shows that roughly about 50% of the config changes are very small one-line or two-line changes. On the other hand, large config changes are not a negligible fraction. 8.7% of the updates to compiled config modify more than 100 lines of the JSON files.

How many co-authors update the same config? Table 3 shows that most configs are only updated by a small number of co-authors. Specifically, 79.6% of the compiled configs are only updated by one or two authors, whereas 91.5% of the raw configs are only updated by one or two authors. It is more skewed for raw configs, because most raw configs are updated by automation tools, which are counted as a single author. On the other hand, some configs are updated by a large number of co-authors. For example, there is a sitevar updated by 727 authors over its two years of lifetime. For future work, it would be helpful to automatically flag high-risk updates on these highly-shared configs.

Is a config's co-authorship pattern significantly different from that of regular C++/Python/Java code? No. This is because Facebook adopts the DevOps model, where engineers developing a software feature also do the configuration-related operational work in production. The "fbcode" column in Table 3 shows the breakdown for Facebook's backend code repository, which is primarily C++, Python, and Java. There is no big difference between the "compiled config" column and the "fbcode" column. One subtle but im-

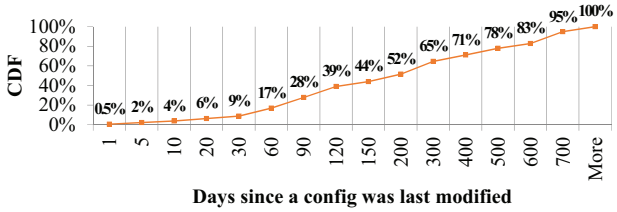

Figure 9: Freshness of configs.

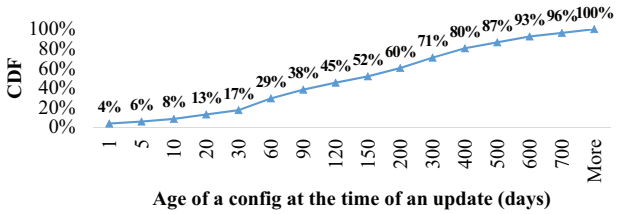

Figure 10: Age of a config at the time of an update.

No. of updates in lifetime	Compiled config	Raw config
1	**25.0%**	56.9%
2	24.9%	23.7%
3	14.1%	5.2%
4	7.5%	3.2%
[5, 10]	15.9%	6.6%
[11, 100]	11.6%	3.0%
[101, 1000]	0.8%	0.7%
[1001, ∞)	0.2%	0.7%

Table 1: Number of times that a config gets updated. How to read the bold cell: **25.0%** of compiled configs are written only once, i.e., created and then never updated.

No. of line changes in a config update	Compiled config	Source code	Raw config
1	2.5%	2.7%	2.3%
2	**49.5%**	44.3%	48.6%
[3, 4]	9.9%	13.5%	32.5%
[5, 6]	3.9%	4.6%	4.2%
[7, 10]	7.4%	6.1%	3.6%
[11, 50]	15.3%	19.3%	5.7%
[51, 100]	2.8%	2.3%	1.1%
[101, ∞)	8.7%	7.3%	2.0%

Table 2: Number of line changes in a config update. How to read the bold cell: **49.5%** of the updates to compiled configs are two-line changes.

No. of co-authors	Compiled config	Raw config	fbcode
1	**49.5**%	70.0%	44.0%
2	30.1%	21.5%	37.7%
3	9.2%	5.1%	7.6%
4	3.9%	1.4%	3.6%
[5, 10]	5.7%	1.2%	5.6%
[11, 50]	1.3%	0.6%	1.4%
[51, 100]	0.2%	0.1%	0.02%
[101, ∞)	0.04%	0.002%	0.007%

Table 3: Number of co-authors of configs. How to read the bold cell: **49.5%** of the compiled configs have a single author throughout their lifetime.

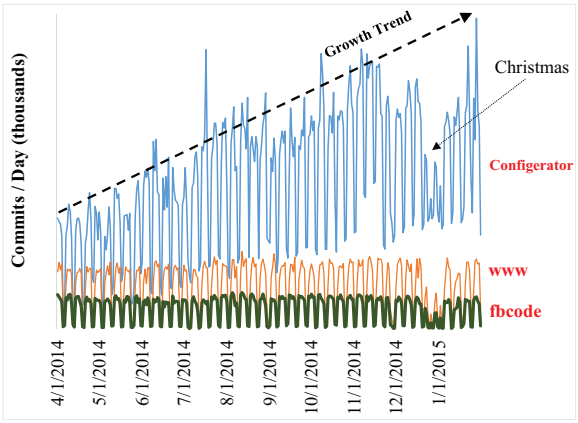

Figure 11: Daily commit throughput of repositories.

portant difference is at the high end. 0.24% of the compiled configs has more than 50 co-authors, whereas only 0.027% of the files in fbcode has more than 50 co-authors. The relative difference is large.

6.3 Configerator & Gatekeeper Performance

Figure 11 compares Configerator's daily commit throughput (i.e., the number of times code/configs are checked into the git repository) with those of *www* (frontend code repository) and *fbcode* (backend code repository). (Facebook has other code repositories not shown here.) Figure 11 highlights the scaling challenge we are facing. In 10 months, the peak daily commit throughput grows by 180%. The periodic peaks and valleys are due to the weekly pattern, i.e., less commits on weekends. Configerator has a high commit throughput even on weekends, because a significant fraction of config changes are automated by tools and done continuously. Specifically, Configerator's weekend commit throughput is about 33% of the busiest weekday commit throughput, whereas this ratio is about 10% for www and 7% for fbcode.

Figure 12 shows Configerator's hourly commit throughput in a week. It exhibits both a weekly pattern (low activities during the weekend) and a daily pattern (peaks between 10AM-6PM). There are a steady number of commits throughout the nights and weekends. Those are automated commits, which account for 39% of the total commits.

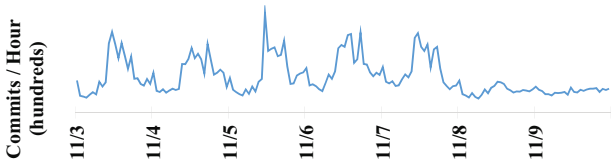

Figure 12: Configerator's hourly commit throughput.

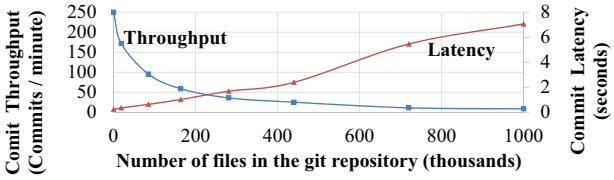

Figure 13: Configerator's maximum commit throughput (measured from a sandbox instead of production).

Figure 13 shows Configerator's maximum commit throughput as a function of the git repository size. It is measured from a sandbox setup under a synthetic stress load test. The git repository is built up by replaying Configerator's production git history from the beginning. To go beyond Configerator's current repository size, we project the repository's future growth by generating synthetic git commits that follow the statistical distribution of past real git commits. Figure 13 shows that the commit throughput is not scalable with respect to the repository size, because the execution time of many git operations increases with the number of files in the repository and the depth of the git history. The right y-axis is simply $latency = \frac{60}{throughput}$, which makes the trend more obvious. This "*latency*" is just the execution time excluding any queueing time. To improve throughput and reduce latency, Configerator is in the process of migration to multiple smaller git repositories that collectively serve a partitioned global name space (see Section 3.6).

When an engineer saves a config change, it takes about ten minutes to go through automated canary tests. This long testing time is needed in order to reliably determine whether the application is healthy under the new config. After canary tests, how long does it take to commit the change and propagate it to all servers subscribing to the config? This latency can be broken down into three parts: 1) It takes about 5 seconds to commit the change into the shared git repository, because git is slow on a large repository; 2) The git tailer (see Figure 3) takes about 5 seconds to fetch config changes from the shared git repository; 3) The git tailer writes the change to Zeus, which propagates the change to all subscribing servers through a distribution tree. The last step takes about 4.5 seconds to reach hundreds of thousands of servers distributed across multiple continents.

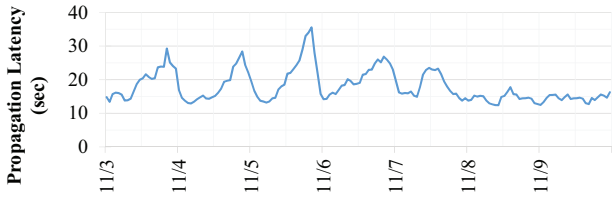

Figure 14: Latency between committing a config change and the new config reaching the production servers (the week of 11/3/2014).

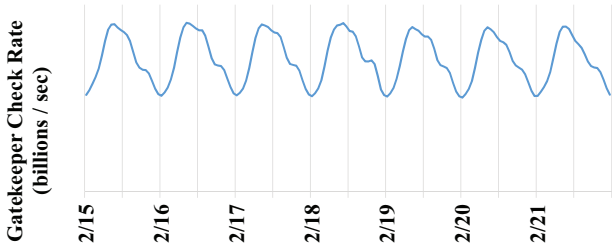

Figure 15: Gatekeeper check throughput (the week of 2/15/2015).

Figure 14 shows the end-to-end latency of the three steps above. It is measured from production. The baseline latency is about 14.5 seconds, but it increases with the load. It shows both a daily pattern (due to the low load at nights) and a weekly pattern (due to the low load on weekends). The major challenge in reducing the latency is to speed up git operations on a large repository. On the other hand, latency is less critical for Configerator, because automated canary tests take about ten minutes anyway.

Gatekeeper projects manage product feature rollouts. When a user accesses *facebook.com*, the Gatekeeper projects are checked in realtime to determine what features to enable for the user. Figure 15 shows the total number of Gatekeeper checks across the site. Because the check throughput is high (billions of checks per second) and some Gatekeeper restraints are data intensive, currently Gatekeeper consumes a significant percentage of the total CPU of the frontend clusters that consist of hundreds of thousands of servers. We constantly work on improving Gatekeeper's efficiency. On the other hand, we consider this "overhead" worthwhile, because it enables Facebook engineers to iterate rapidly on new product features. This is evidenced by the popularity of Gatekeeper. In 2014, tens of thousands of Gatekeeper projects were created or updated to actively manage the rollouts of a huge number of micro product features.

6.4 Configuration Errors

To illustrate the challenges in safeguarding configs, we discuss several real examples of configuration errors as well as a statistical analysis of configuration errors.

The first example is related to the config rollout process. When both the client code and the config schema are updated together, they may not get deployed to a server at the same time. Typically, the new client code can read the old config schema, but the old client code cannot read the new config schema. The latter was what happened in an incident. An engineer changed both the client code and the config schema. She checked in the client code and thought it would be included in the code deployment happening later on that day. However, the release branch was cut earlier and her new code unknowingly missed that release. Five days later, the engineer committed a new config using the new schema. The automated canary testing tool initially only deployed the new config to 20 servers in production, and monitored their health. It compared the error logs of those 20 servers with those of the rest of the production servers, and detected a log spew, i.e., rapid growth of error logs. It aborted the rollout and prevented an outage.

Another incident was less lucky. An engineer made a config change, which was rejected by the automated canary tool, because it caused some instances of the application to crash. The engineer stared at the config change. It seemed such a trivial and innocent change that nothing could possibly go wrong. "*It must be a false positive of the canary tool!*" She overrode the tool's rejection and deployed the config, which caused more crashes. She mitigated the problem by immediately reverting the config change. It turned out that the config change itself was indeed correct, but it caused the application to exercise a new code path and triggered a subtle race-condition bug in the code. This incident highlights that problems could arise in any unexpected ways.

Enhancing automated canary tests is a never ending battle, as shown in the example below. An engineer introduced a configuration error that sent mobile requests down a rare code path to fetch data from a backend store. It put too much load on the data store and dramatically increased the latency. At that time, automated canary tests did not catch the problem, because the testing was done on a limited number of servers and the small scale testing was insufficient to cause any load issue. The latency increase became evident only after the config was deployed site wide. Since then, we added a canary phase to test a new config on thousands of servers in a cluster in order to catch cluster-level load issues.

In addition to the examples above, we manually analyzed the high-impact incidents during a three-month period. We found that 16% of the incidents were related to configuration management, while the rest were dominated by software bugs. The table below shows the breakdown of the configuration-related incidents.

Type of Config Issues	*Percentage*
Type I: common config errors	42%
Type II: subtle config errors	36%
Type III: valid config changes exposing code bugs	22%

Type I errors are obvious once spotted, e.g., typos, out-of-bound values, and referencing an incorrect cluster. They can benefit from more careful config reviews. Type II errors are harder to anticipate ahead of time, e.g., load-related issues, failure-induced issues, or butterfly effects. The root causes of type III issues are actually in the code rather than in the configs. The high percentage of type III issues was a surprise to us initially. All types of config issues can benefit from better canary testing, which is a focus of our ongoing work.

6.5 Operational Experience

Facebook's configuration management team adopts the DevOps model. A small group of engineers (i.e., the authors) are responsible for 1) implementing new features and bug fixes, 2) deploying new versions of the tools into production, 3) monitoring the health of the tools and resolving production issues, and 4) supporting the engineering community, e.g., answering questions and reviewing code related to the use of the tools. Everyone does everything, without separation of roles such as architect, software engineer, test engineer, and system administrator. The small team is highly leveraged, because we support thousands of engineers using our tools and manage the configs on hundreds of thousands of servers and more than one billion mobile devices.

Engineers on the team rotate through an oncall schedule. The oncall shields the rest of the team to focus on development work. Monitoring tools escalate urgent issues to the oncall through automated phone calls. The tool users post questions about non-urgent issues into the related Facebook groups, which are answered by the oncall when she is available, or answered by users helping each other. We strive to educate our large user community and make them self-sufficient. We give lectures and lab sessions in the bootcamp, where new hires go through weeks of training.

Although Configerator's architecture in Figure 3 seems complex, we rarely have outages for the Configerator components, thanks in part to the built-in redundancy and automated failure recovery. On the other hand, there are more outages caused by configuration errors breaking products, and we strive to guard against them, e.g., by continuously enhancing automated canary tests.

The Configerator proxy is the most sensitive component, because it runs on almost every server and it is hard to anticipate all possible problems in diverse environments. Many product teams enroll their servers in a testing environment to verify that a new proxy does not break their products. The proxy rollout is always done carefully in a staged fashion.

6.6 Configuration Management Culture

An Internet service's configuration management process and tools reflect the company's engineering culture. At the conservative extreme, all config changes must go through a central committee for approval and are carefully executed by a closed group of operation engineers. At the moving-fast extreme, every engineer can change any config directly on the site and claim "*test it live!*", which is not uncommon in startups. The authors experienced both extremes (as well as something in-between) at different companies.

Over the years, Facebook has evolved from optional diff review and optional manual testing for config changes, to mandatory diff review and mandatory manual testing. We also put more emphasis on automated canary tests and automated continuous integration tests. The tools do support access control (i.e., only white-listed engineers can change certain critical configs), but that is an exception rather than the norm. We empower individual engineers to use their best judgment to roll out config changes quickly, and build various automated validation or testing tools as the safety net. We expect Facebook's configuration management culture to further evolve, even in significant ways.

Facebook engineers iterate rapidly on product features by releasing software early and frequently. This is inherent to Facebook's engineering culture. It requires tools to manage product rollouts and run A/B tests to identify promising product features. Gatekeeper and other A/B testing tools are developed as the first-class citizens to support this engineering culture, and are widely used by product engineers.

7. Related Work

There is limited publication on runtime configuration management for Internet services. The closest to our work is Akamai's ACMS system [30]. ACMS is a configuration storage and distribution system, similar to the Zeus component in our stack. ACMS uses the pull model whereas Zeus uses the push model. Moreover, the scope of our work is much broader, including configuration authoring and validation, code review, version control, and automated testing. Our use cases are also much broader, from incremental product launch to testing parameters on mobile devices. In terms of config distribution, just on the server side, our system is 10-100 times larger than ACMS; counting in mobile devices, our system is 10,000-100,000 times larger.

Configuration management is an overloaded term. It has been used to mean different things: 1) controlling an application's runtime behavior; 2) source code version control [9]; and 3) software deployment, e.g., Chef [7], Puppet [28], Autopilot [19], and other tools covered in the survey [10]. ACMS and our tools fall into the first category.

"*Configuration-as-code*" has different forms of realization. Chef [7] executes code on the target server at the deployment time, whereas Configerator executes code during the development phase and then pushes the JSON file to all servers during the deployment phase. The difference stems from their focuses: software installation vs. managing runtime behavior. Configerator indeed has some special use cases where scripts are stored as raw configs and pushed to servers for execution, but those are very rare.

Chubby [6] and ZooKeeper [18] provide coordination services for distributed systems, and can be used to store application metadata. We use Zeus, an enhanced version

ZooKeeper, to store configs, and use observers to form a distribution tree. Thialfi [1] delivers object update notifications to clients that registered their interests in the objects. It potentially can be used to deliver config change notifications.

Oppenheimer et al. [24] studied why Internet services fail, and identified configuration errors as a major source of outages. Configuration error is a well studied research topic [17, 21, 25, 33, 35–37]. Our focus is to prevent configuration errors through validators, code review, manual tests, automated canary tests, and automated integration tests. Configuration debugging tools [3, 31, 32] are complementary to our work. We can benefit from these tools to diagnose the root cause of a configuration error. Spex [34] infers configuration constraints from software source code. It might help automate the process of writing Configerator validators.

Like LBFS [23] and DTD [22], MobileConfig uses hash to detect and avoid duplicate data transfer. It is unique in that legacy versions of an app may access the same config using different schemas and need to fetch different data.

Configerator not only uses git for version control, but also uses a git push to trigger an immediate config deployment. Many Cloud platforms [4, 15, 16] adopt a similar mechanism, where a git push triggers a code deployment.

Our tools follow many best practices in software engineering, including code review, verification (in the form of validators), continuous integration [12], canary test, and deployment automation. Our contribution is to apply these principles to large-scale configuration management.

8. Conclusion

We presented Facebook's configuration management stack. Our main contributions are 1) defining the problem space and the use cases, 2) describing a holistic solution, and 3) reporting usage statistics and operational experience from Facebook's production system. Our major future work includes scaling Configerator, enhancing automated canary, expanding MobileConfig to cover more apps, improving the config abstraction (e.g., introducing config inheritance), and flagging high-risk config updates based on historical data.

The technology we described is not exclusive for large Internet services. It matters for small systems as well. Anecdotally, the first primitive version of the Sitevars tool was introduced more than ten years ago, when Facebook was still small. What are the principles or knowledge that might be applied beyond Facebook? We summarize our thoughts below, but caution readers the potential risk of overgeneralization.

- Agile configuration management enables agile software development. For example, gating product rollouts via config changes reduces the risks associated with frequent software releases. A/B testing tools allow engineers to quickly prototype product features and fail fast. Even if these specific techniques are not suitable for your organization, consider other dramatic improvements in your configuration management tools to help accelerate your software development process.
- With proper tool supports, it is feasible for even a large organization to practice "*open*" configuration management, i.e., almost every engineer is allowed to make online config changes. Although it seems risky and might not be suitable for every organization, it is indeed feasible and can be beneficial to agile software development.
- It takes a holistic approach to defend against configuration errors, including config authoring, validation, code review, manual testing, automated integration tests, and automated canary tests.
- Although the use cases of configuration management can be very diverse (e.g., from gating product rollouts to A/B testing), it is feasible and beneficial to support all of them on top of a uniform and flexible foundation, with additional tools providing specialized functions (see Figure 1). Otherwise, inferior wheels will be reinvented. At Facebook, it is a long history of fragmented solutions converging onto Configerator.
- For nontrivial configs, it is more productive and less error prone for engineers to write programs to generate the configs as opposed to manually editing the configs.
- In data centers, it is more efficient to use the *push model* to distribute config updates through a tree.
- The hybrid *pull-push model* is more suitable for mobile apps, because push notification alone is unreliable.
- Separating the distribution of a large config's small metadata from the distribution of its bulk content (through a P2P protocol) makes the solution scalable, without sacrificing the data consistency guarantee.
- A typical git setup cannot provide sufficient commit throughput for large-scale configuration management. The solution is to use multiple git repositories to collectively serve a partitioned global name space, and delegate commits to the landing strip to avoid contention.
- The config use cases and usage statistics we reported may motivate future research. For example, our data show that old configs do get updated, and many configs are updated multiple times. It would be helpful to automatically flag high-risk updates based on the past history, e.g., a dormant config is suddenly changed in an unusual way.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Shan Lu, for their valuable feedback. The authors are the current or the most recent members of Facebook's configuration management team. In the past, many other colleagues contributed to Configerator, Gatekeeper, Sitevars, and Zeus. We thank all of them for their contributions. We thank our summer intern, Peng Huang, for writing scripts to replay the git history used in Figure 13.

References

[1] ADYA, A., COOPER, G., MYERS, D., AND PIATEK, M. Thialfi: a client notification service for internet-scale applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 129–142. SOSP'11.

[2] APACHE THRIFT. http://thrift.apache.org/.

[3] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation* (2010), pp. 237–250. OSDI'10.

[4] AWS ELASTIC BEANSTALK. http://aws.amazon.com/elasticbeanstalk/.

[5] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., ET AL. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013). USENIX ATC'13.

[6] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 335–350. OSDI'06.

[7] CHEF. http://www.opscode.com/chef/.

[8] COHEN, B. Incentives build robustness in bittorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer systems* (2003), pp. 68–72.

[9] CONRADI, R., AND WESTFECHTEL, B. Version models for software configuration management. *ACM Computing Surveys 30*, 2 (1998), 232–282.

[10] DELAET, T., JOOSEN, W., AND VAN BRABANT, B. A survey of system configuration tools. In *Proceedings of the 24th Large Installation System Administration Conference* (2010). LISA'10.

[11] DIBOWITZ, P. Really large scale systems configuration: config management @ Facebook, 2013. https://www.socallinuxexpo.org/scale11x-supporting/default/files/presentations/cfgmgmt.pdf.

[12] DUVALL, P. M., MATYAS, S., AND GLOVER, A. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[13] FOWLER, M., AND HIGHSMITH, J. The agile manifesto. *Software Development 9*, 8 (2001), 28–35.

[14] GIT. http://git-scm.com/.

[15] GOOGLE APP ENGINE. https://appengine.google.com/.

[16] HEROKU. https://www.heroku.com/.

[17] HUANG, P., BOLOSKY, W. J., SINGH, A., AND ZHOU, Y. ConfValley: A systematic configuration validation framework for cloud services. In *Proceedings of the 10th European Conference on Computer Systems* (2015), p. 19. EuroSys'15.

[18] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), pp. 11–11. USENIX ATC'10.

[19] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review 41*, 2 (2007), 60–67.

[20] JAVASCRIPT OBJECT NOTATION (JSON). http://www.json.org/.

[21] MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2002), pp. 3–16. SIGCOMM'02.

[22] MOGUL, J. C., CHAN, Y.-M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (2004), pp. 43–56. NSDI'04.

[23] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems principles* (2001), pp. 174–187. SOSP'01.

[24] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (2003). USITS'03.

[25] PAPPAS, V., XU, Z., LU, S., MASSEY, D., TERZIS, A., AND ZHANG, L. Impact of configuration errors on DNS robustness. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2004), pp. 319–330. SIGCOMM'04.

[26] PHABRICATOR. http://phabricator.org/.

[27] POWER, A., 2011. Making Facebook Self-Healing, https://www.facebook.com/notes/facebook-engineering/making-facebook-self-healing/10150275248698920.

[28] PUPPET. https://puppetlabs.com/.

[29] ROSSI, C. Ship early and ship twice as often. https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920.

[30] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. ACMS: the Akamai configuration management system. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (2005), pp. 245–258. NSDI'05.

[31] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (2007), pp. 237–250. SOSP'07.

[32] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation* (2004), pp. 77–90. OSDI'04.

[33] WOOL, A. A quantitative study of firewall configuration errors. *IEEE Computer 37*, 6 (2004), 62–67.

[34] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), pp. 244–259. SOSP'13.

[35] XU, T., AND ZHOU, Y. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys 47*, 4 (2015), 70.

[36] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 159–172. SOSP'11.

[37] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th Architectural Support for Programming Languages and Operating Systems* (2014), pp. 687–700. ASPLOS'14.