Defcon: Preventing Overload with Graceful Feature Degradation

Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard, *Meta Platforms, Inc.*

https://www.usenix.org/conference/osdi23/presentation/meza

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Defcon: Preventing Overload with Graceful Feature Degradation

Justin J. Meza Thote Gowda Ahmed Eid Tomiwa Ijaware Dmitry Chernyshev
Yi Yu Md Nazim Uddin Rohan Das Chad Nachiappan Sari Tran Shuyang Shi
Tina Luo David Ke Hong Sankaralingam Panneerselvam Hans Ragas
Svetlin Manavski Weidong Wang Francois Richard

Meta Platforms, Inc.

Abstract

Every day, billions of people depend on Internet services for communication, commerce, and entertainment. Yet planetary-scale data center infrastructures consisting of millions of servers experience unplanned capacity outages and unexpected demand for resources; how can such infrastructures remain reliable in the face of capacity and workload flux?

In this paper, we introduce Defcon, a system for improving the availability of large-scale, globally-distributed Internet services using graceful *feature* degradation. In response to overload conditions, Defcon enables site operators to gradually disable less-critical features in order to reduce resource demand. Defcon presents a common interface to product developers to define feature *knobs* that represent degradation capabilities. Defcon automatically tests knobs to understand each knob's product- and infrastructure-level trade-offs. At Meta, we have used Defcon to improve global product availability in the face of worldwide demand-surges in addition to large-scale infrastructure failures.

1 Introduction

Large-scale, globally-distributed Internet services, such as those operated by Alibaba, Amazon, Google, Meta, Microsoft, and Netflix, power modern human life by providing access to communication, commerce, entertainment, and many other experiences. At the same time, rapid advances in finance, artificial intelligence, machine learning, and virtual/augmented reality have solidified the utility of Internet services for much of humanity for the foreseeable future.

Internet services consist of *features* – functional building blocks that make up a larger product. For example, a video product consists of a search feature, a playback feature, a recommendation feature, and so on. Features are hierarchical: A *top-level* playback feature may itself consist of a video quality feature and a closed-caption feature, for example. Features, and the *products* they make up, power Internet services.

Products (and, by extension, features) run in data centers distributed around the planet. Analogous to the familiar von Neumann architecture, computing at a planetary scale requires input/output (in the form of HTTP and RPC requests), computation (in the form of front-end servers), interconnect (the network backbone), caching, storage, and so on. Site operators deploy these resources within geographically-distributed data centers with the goal of ensuring that the workload *demanded* by users does not exceed the resources *supplied* by the network, servers, and so on.

Planning data center resources well requires predicting the future – or at least trying to. Capacity engineers rely on detailed user demand forecasts and server supply models to decide how and where to purchase and deploy resources, but alas, prophesy yet remains elusive: Errors and inaccuracy creep into models and forecasts, making data center capacity planning at times more of an art than a science. In addition, unpredictable world events – like global pandemics – can render even the most sophisticated predictions obsolete overnight.

At the end of the day, the people that use Internet services care about *availability*: Can they use the product that they want to use when they want to use it? Toward that end, companies work hard to ensure their products remain highly available. But what happens when things do not go according plan, such as during a persistent product demand increase due to a global pandemic, or when recovering from a global outage? Can we achieve high product availability without sacrificing additional resources? Can we be more efficient for rare – but inevitable – partial outages and survive them without additional server resources?

For example, Figure 1 shows a *real-world* surge in demand for one of Meta's products, Facebook (measured on the y-axis in mega-instructions per second, or MIPS, executed by front-end servers for the product), that occurred over several hours on October 27, 2022. Localized peaks toward the left and right of the chart illustrate software deployment on the front-end systems, which consumes additional resources due to idle hosts updating their binaries and cold cache effects – these are expected behaviors. At around 5AM PDT, however, an unexpected increase in demand for the product happened to coincide with the daily peak usage of the product (shown

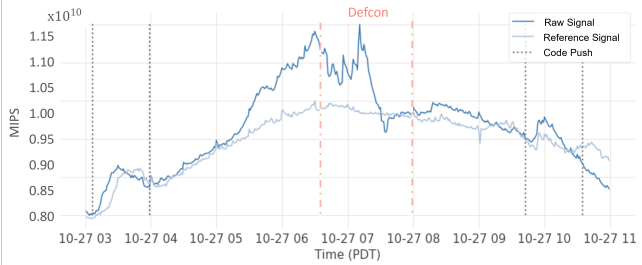

Figure 1: Defcon in action during a real-world site event. *(See Section 1 for explanation.)*

from a previous day as "reference signal") leading to constructive interference and causing the product to run out of capacity and dangerously approach an *overload* condition (approximately 1.15×10^{10} MIPS) at which point fail-slow behavior and overload would occur. At around 6:35AM PDT (vertical dotted line), site operators engaged a system, which we present in this paper, in order to safely and efficiently reduce resource consumption (MIPS), *while still preserving access to core product functionality for all users*, avoiding an outage. Around 7:15AM PDT, as demand for the product continued to increase, site operators further engaged a next *level* of the system – leading to a correspondingly larger decrease in resource consumption, *bending the traffic (MIPS) surge curve* to restore it to nominal amounts of resource demand. After the surge had passed, at around 8:00AM PDT, site operators disengaged the system, restoring the product's features to their original state.

In this paper, we present *Defcon*, a system to provide graceful *feature* degradation for Internet services. Defcon achieves high availability without sacrificing additional server resources by allowing site operators to *dynamically turn off product features* in response to rare (e.g., monthly or yearly) demand spikes or even unpredictable product demand increases. The key insight of Defcon is that *not all product features provide equal value* – many features can safely be turned off for short periods of time without altering a product's fundamental behavior. Human guidance is used to define and actuate "knobs" – control flow annotations that represent the best capacity savings and user experience trade-offs.

We characterize the overload problem and solution space, apply a rigorous data-scientific methodology to analyze knob behavior, and describe a real-world at-scale testing methodology to validate the efficacy of Defcon. We also shed light on the design and organization of large-scale, real-world systems from the field as our approach accurately reflects the trade-offs involved in designing and implementing an initial solution to an emerging problem under realistic constraints. A key contribution of this paper is to prove the efficacy of feature degradation to help solve the overload problem in distributed systems.

We describe the design and implementation of Defcon and our experience operating Defcon in production over the course of three years. We evaluate our approach using a combination of continual at-scale controlled tests as well as case studies from production incidents, including during a sustained demand surge. Overall, we find graceful feature degradation to be a powerful design pattern for system architects to efficiently improve the availability of large-scale distributed systems.

2 Background

Graceful degradation pervades the natural world: Removing ballast to prevent a ship from capsizing, escalators losing power and becoming ordinary stairs, starfish reproducing a lost limb, and so on. We observe analogous patterns of graceful degradation in the realm of computing and provide a brief overview of these techniques as well as a backdrop for why graceful degradation matters in large-scale Internet services, next.

2.1 Data Center Capacity Management

Modern hyper-scale data center infrastructures rely on server *capacity* distributed across the planet in order to support the diverse resource needs of the services that run in the data centers. Capacity Engineers rely on two inputs in order to make data center capacity planning decisions: *workload resource demand* and *server resource supply*.

Workload resource demand models the resource needs of a product in order to support its set of features. Capacity engineers normalize resources to a common unit baseline in order to plan resources across different server architectures or generations (e.g., Relative Resource Units, or RRUs) where resource types include computational throughput, storage capacity, memory bandwidth, and network bandwidth. Modeling workload resource demand involves understanding how many RRUs of different resource types are required to support product features. To accurately model future resource demand, engineers scale current resource demand based on feature growth projections.

Of course, in reality, resource supply and demand can behave in unpredictable ways. For example, a workload pattern change can change resource demand, while a data center outage can decrease resource supply. A key challenge, therefore, is *how to allocate resources in the face of constant infrastructure and workload flux*[1]. In many traditional systems, scenarios where resource demand > resource supply leads to fail-slow – and, eventually, overload-induced – system unavailability.

[1]Note that systems in Meta's infrastructure are already equipped to automatically scale up and down capacity in response to predictable (e.g., diurnal) demand changes. Even so, there still comes a point when there is no remaining capacity to elastically expand a service into (such as during unpredictable load spikes or large outages).

Potential Solution	Description	Additional Resources	Engineering Effort	User Impact
0. Do Nothing	Allow overload to happen, leading to product outages.	None	None	Very High
1. Overprovision Resources	Increase server resources, leading to lower steady state utilization. Cannot fully predict future traffic patterns.	Prohibitively High	None	Potentially None
2. Drop User Requests	Reduce work by discarding user requests at a load balancer level (L4 or L7) before they enter into a data center.	None	Medium	High
3. Shed Server Load	Modify micro-services to decide when and which requests to drop for their service.	None	High	Medium
4. Degrade Product Features	Annotate control flow and avoid executing certain features on-demand.	None	High	Low

Table 1: Potential solutions to the overload problem and their associated trade-offs.

2.2 The Overload Problem

System *overload* occurs when the requested throughput of a service (e.g., measured in queries per second, QPS) exceeds the capabilities of the system, leading to a phenomena known as *congestion collapse* whereby *goodput* (a measure of the rate of successful responses from the service), decreases [2, 38]. Systems of any size can become overloaded, but the overload problem is especially acute in large, geographically-distributed Internet services, which can cause cascading failure scenarios, and can lead to widespread outages [12]. *Overload remains a fundamental problem in the operation of distributed systems.*

Meta's infrastructure is organized around a collection of geographically-distributed data center failure domains, each representing around 5–12% of the overall capacity. Common failures such as bugs, network/power outages, and incorrect configuration happen within these failure domains and we have found 5–20% of savings to be a sweet spot for capacity savings for mitigating the risk of cascading failures due to overload. In this work, we assume a baseline of an overload-induced metastable failure state that leads to product outages for large portions of users for minutes to hours at a time.

Table 1 summarizes some potential solutions to the overload problem and how they trade off the amount of hardware resources (Server Resources), the amount of effort required of engineers to implement and maintain (Engineering Effort), and the potential impact to users (User Impact). For example, one way to attempt to solve the overload problem is to simply allocate more server resources for a distributed system (option 1). While potentially effective, simply allocating more resources can be inefficient, leading to low resource utilization when traffic is not at its infrequent (e.g., on the order of months or years) projected peak.

Furthermore, we can never perfectly predict traffic patterns and real-world events can often thwart even the best preparations. Take the COVID-19 pandemic as an example: In 2020, as more persons began to shelter in place, communication that was once in-person began shifting to occur online. Figure 2 shows an example of how traffic for one product at Meta greatly exceeded its pre-pandemic resource plans. During global crises, Internet services often become more important than ever for humans to communicate and remain connected with each other. And while at Meta we were able to survive the COVID-19 demand surge, we wondered: *"Can we build systems that are inherently resilient in the face of unforeseen overload?"*

To answer this question, we found options 2–4 compelling. Note that options 2 and 3 both reduce work, but whereas option 2 reduces work at its *source*, option 3 reduces work at its *destination*. Specifically, for option 3, we considered a fine-grained backpressure-based approach, which led to noticeable user impact when capacity demand exceeded capacity supply and requests could not be processed. After evaluating the potential trade-offs at Meta, we opted for a technique to minimize user impact and found option 4, Degrade Features, to achieve the best trade-off: No additional server resources and low impact to users, albeit with an investment in engineering effort (which we qualify in Section 5).

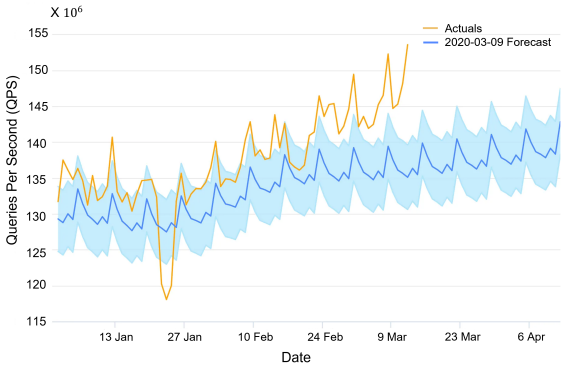

Figure 2: Real-world events can often thwart event the most sophisticated preparation techniques, as shown by this graph of actual versus forecasted demand for Facebook in 2020 during the onset of the COVID-19 pandemic.

2.3 Related Work

Degradation is a self-adaptation technique that reduces the amount of work that servers need to perform to stay resilient during resource shortages, spiky load, and reduced hardware performance [12] that would otherwise cause a distributed system to enter a failure state [6, 16]. Degradation has been considered in many different contexts of system design, such as storage systems [8, 22, 31, 39], processor design [4, 17], cloud computing infrastructure [9, 11, 20, 25, 29], edge computing systems [21], web server applications [1, 2, 14], search engines [10], and mail services [27]. In network systems, graceful degradation is used to handle overload of network resources through intelligent connection management [36], traffic prioritization [3], traffic handover control [5], security hardening [15, 34], as examples.

Degrading the static content of a website was proposed in [2] and relevant techniques have been extended to dynamic content [26]. Degradation has also been proposed by cloud Infrastructure as a Service (IaaS) providers as a feature to increase cloud utilization [29] and used for adapting to the high network variability and possible network disruptions in edge computing infrastructure [21]. Defcon contributes to this area of research by developing a product-level, *feature-centric* framework to perform configurable *graceful* degradation of large-scale geo-distributed micro-services during spiky load and disaster events and providing real-world insights on how to build and operate such a system from its global of deployment in products at a large scale.

One alternative approach for surviving resource shortages during load spike or outage events is load shedding [11,37,38], which drops a proportion of load by dropping request traffic when a server approaches overload. However, load shedding sacrifices availability guarantees and broadly impacts user experience. In contrast, degradation techniques aim to provide high availability of products and services to users around the globe, which is critical to minimize impact.

Another area of related research is on specifying and realizing degradation for distributed systems. A relaxation lattice method was proposed for specifying the behavior of degradation [13]. Furthermore, specifications and implementations of degradation were presented in [39] as a complementary mechanism to fault tolerance in the design of highly-available distributed systems. Availability Knob [30] was proposed to provide a variety of availability guarantees, improving the utilization of reliability-heterogeneous infrastructures. In this work, we adopt the "knob" nomenclature, although for different means. Whereas Availability Knobs specify availability SLA flexibility, knobs as used in this work describe source code control flow annotations which can be enabled or disabled at-will.

Client-managed degradation was explored in the context of features like low power modes [18]. Our approach differs from client-managed degradation in three key ways. First, in contrast to an ad-hoc approach to define individual points of degradation in client code (which, like a low power mode, then effectively become new "features" to maintain in the client), our approach provides a framework (knobs) that developers can use to efficiently encapsulate existing features, significantly reducing the development cost of degradation. Second, our approach provides developers with a framework to automatically test, analyze knob savings, and manage the lifecycle of knobs. Third, our approach extends to both client-side and server-side knobs, as it abstracts the knob control plane into configuration management as opposed to custom client (or server) code.

2.4 Graceful Feature Degradation

In this work, we ask the question, "Can we design distributed systems that remain available even when resource demand $>$ resource supply?" While such systems would violate traditional system design assumptions, our key insight is that *not all features of a product are equally important* – if we can identify *essential* features (such as the ability to send a message in a messaging product) versus *fungible* features (such as an online status indicator for whether the message recipient is currently online), then we can gracefully *trade off fungible features for on-demand server resources*, while still preserving *essential features*. For example, the number of results can be considered as an adjustable *feature* for a search product.

In this paper, we introduce the qualifier graceful *feature* degradation to refer to the property of a large, globally-distributed system to dynamically modify its behavior (features) in order to dynamically (i.e., *at runtime, without* recompilation or changing binary flags) alter its control flow for the purpose of reducing the system's resource requirements. From here on, we use the terms "graceful degradation" and "graceful feature degradation" interchangeably.

3 Defcon

Defcon is a system to implement graceful degradation in large-scale distributed systems. Defcon is designed to be used during infrequent site emergency situations where demand is greater than supply. There are many reasons why a system may encounter situations where demand for the system's resource exceeds the supply of resources for the system. Some examples are data center outages, load spikes during special events like New Year's Eve, service overload due to a bug in a software deployment, and so on. We provide an overview of Defcon and discuss the design and implementation of its key system components, next.

3.1 Overview

Figure 3 shows an overview of Defcon. *Knobs* annotate program control flow eligible for degradation and follow a well-defined API. Product engineers use a Knob Definition Framework to annotate source code that can be degraded. These knobs are controlled using a Knob Actuator Service by site operators according to a policy. Not shown in the figure, a *Knob Testing Framework* registers knobs defined in the code base and automatically tests them to understand the sensitivity of product experience and resource consumption when the knob is turned on. We discuss each of these components next.

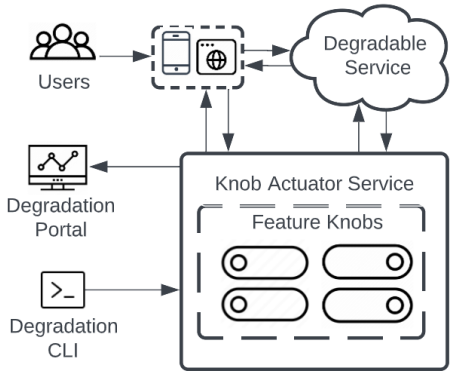

Figure 3: Overview of Defcon system architecture. Product software engineers use a *Knob Definition Framework* to label sections of control flow corresponding to specific features to conditionally execute. A *Knob Actuator Service* controls which knobs (and their corresponding features) are enabled or disabled in response to real-time events. Mobile clients, web clients, and micro-services all communicate via configuration with the Knob Actuator Service to dynamically determine control flow. A Degradation Portal provides insights for site operators to understand which knobs to enable in response to server resource shortages and a Degradation CLI allows humans to rapidly control knobs *en masse*.

3.2 Knob Definition Framework

A *knob* is a switch to enable or disable a feature in the code. Unlike feature flags, which require a binary restart in order to take effect, knobs are controlled dynamically while a binary is running. Knobs are implemented by a client library (or sidecar service) that determines the current state of each knob and are controlled using a configuration management system [32]. Software developers provide each knob a unique name, which they then can reference in their code. Thus, knobs can span multiple source code files, or even multiple binaries. Knobs come in two flavors:

1. Server-side knobs are implemented in binaries running on the servers in data centers. The advantage of server-side knobs is that we can adjust the knobs' state in seconds without any propagation delays.

2. Client-side knobs are implemented in client code running on phones, tablets, wearables, and so on. The advantage of client-side knobs is that they have the capability to reduce network load by stopping requests sent to the server along side reducing server load due to the request. Client-side knobs can also be controlled conditionally based on device metadata, such as cache state and network bandwidth availability. For example, Meta's mobile apps maintain a client-side cache response freshness threshold value. We update this freshness threshold value during Defcon to control incoming traffic. At Meta, we use server-side configuration to control these values. We use two approaches to propagate knobs state changes to clients, each with its own pros and cons:

2.a. Silent Push Notification (SPN): This approach uses a push notification system to propagate knobs state changes. At Meta, we have large numbers of client devices and the system takes around 30 minutes to finish all push notification jobs to propagate knobs state changes. SNP works like a typical app notification mechanism but instead of showing a notification to a user, the client app updates corresponding configuration fields.

2.b. Mobile Configuration Pull (MCP): In this approach, clients pull updated mobile configurations from servers through an API. At Meta, every client application implements two kinds of configuration-pull mechanisms: (1) A *full configuration pull* happens every 6 hours and pulls updated configuration data for every configuration definition. Full configuration pull is more thorough, but requires more network bandwidth and server resources. (2) During *Emergency Mobile Configuration (EMC) pull*, each client request triggers a server to inspect an emergency configuration file located on the server to fetch updated configuration data for the fields mentioned in the emergency configuration file. EMC consumes less network bandwidth and server resources, but requires manual intervention.

Listing 1 shows an example of defining a knob in Python (although APIs also exist for Rust, C++, Hack [35], and Java). Every knob has a unique name (with a namespace unique for each product name, `Feed` in this example), a *level* corresponding to the magnitude of resource reduction and used for grouping all knobs of a similar magnitude together, and a Boolean enabled state. The `export` statement instructs the build system to generate/update knob source code definitions in the code base.

Listing 1: Knob definition.

```
from configs.knobs import KnobConfig
disableCommentsRanking = KnobConfig(
    name = "Feed/DisableCommentsRanking",
    oncall = "owner_team_oncall",
    level = 2,  # Impact magnitude.
    enabled = True)
export(disableCommentsRanking)
```

Listing 2 shows an example of using a knob in Python. To

use a knob, a developer must inspect the `enabled` field for the knob:[2] If the knob is disabled (the common case), the binary follows its normal control flow; if the knob is enabled (e.g., during an emergency), the binary follows a work-reducing control flow to reduce server resource consumption for every request served.

Listing 2: Knob usage.

```
from configs import ConfigReader
disableCommentsRanking = ConfigReader(
    "Feed/DisableCommentsRanking")
comments = fetchComments()
if (disableCommentsRanking.enabled == False)
    comments.RankUsingModel()
else  # Knob enabled: do less work.
    comments.RankChronologically()
```

At Meta, knobs are not implemented haphazardly, but are instead carefully planned for by product teams with target resource savings set for different knob levels. Even so, the flexibility and ease of knob definition has enabled some products to implement and manage hundreds of knobs. Usually product teams choose the design of their knobs (i.e., server side knobs or client side knobs) based on the product behavior and the trade offs from controlling the demand at different places. Generally, knobs are defined at product feature level to stop the entire control flow across different surfaces.

Defcon knobs are added to both existing and new features. At Meta, features are deployed gradually with server-side controls and experiments. Meta's deployment process requires product engineers to have a single-server side configuration to enable/disable their features. In Meta's infrastructure, features are typically implemented as separate RPCs and therefore there is strong isolation between the control flow of each feature. For shared library code, product engineers have the choice to degrade at the library level or at a finer-grained RPC request level.

This process provides an advantage for developing knobs as a product team can simply extend these feature controls to check for Defcon configuration. Integrating knobs with feature development and deployment processes has other advantages: Ease of running experiments to test a knob for side-effects, measuring the capacity savings from disabling a knob, and measuring the impact of a knob on users (Section 3.4). User impact is then used to classify a knob into the correct Defcon level (Section 3.5). Once a product engineer is satisfied with a knob's behavior, they will explicitly choose to include it in the Defcon system.

To aid product teams in understanding the breadth and behavior of the knobs they have defined, a browser-based graphical user interface is provided to help developers understand target level resource saving expectations, manage knob metadata, visualize knob savings against the target expectations, and understand any user experience trade-offs (using a measurement methodology we discuss later). In turn, emergency responders use this user interface to understand Defcon level savings and the associated impact of enabling knobs.

[2] Knob configuration state is cached within memory on the server a binary is running on and accessed either by a shared library or a sidecar binary, typically requiring no more than microseconds to access.

3.3 Knob Actuator Service

We believe it is important to have a highly reliable tool with minimal dependencies to control Defcon, so that we can use Defcon even when most other systems are unavailable. The Knob Actuator Service is responsible for enabling or disabling (actuating) sets of knobs. Knobs are grouped into three categories: (1) By service name, (2) by product name, and (3) by feature name (such as "search," "video," "feed," and so on).

The Knob Actuator Service also manages metadata for knobs, stored in a geographically-replicated relational (MySQL) database. Knob metadata includes: (1) The engineering oncall responsible for the knob's definition, (2) the engineering team responsible for the knob's usage, and (3) a cache of recent resource and user experience test results (discussed later in this section).

Finally, the Knob Actuator Service is responsible for changing the state of knobs. Knob state changes can be performed for individual knobs or for sets of knobs grouped using one of the three categories (service, product, or feature name). State changes occur in seconds for server-side knobs and in a couple of minutes for client-side knobs (due to the EMC pull cycle duration mentioned before).

While state changes across sets of knobs are used during site events that require additional capacity supply, state changes for *individual* knobs are used for testing knob impact. Knobs can be further selected for only a fraction of users participating in controlled A/B test experiments (discussed in the next sub-section).

At Meta, emergency responders receive notifications for various overload scenarios (including increased demand, decreased capacity, etc.) for services. The emergency responders use a Degradation Policy (defined in Section 3.5) to evaluate if Defcon can and should be used to reduce the load. Once the emergency responders decide on a course of action, they use capacity savings data from recent tests (which are available in a dashboard) to estimate what Defcon level should be enabled, and use the Knob Actuator Service to enable Defcon knobs to reduce the demand to the desired amount.

3.4 Knob Testing Framework

As an emergency response tool, we must test Defcon periodically to ensure its reliability and performance. Since Defcon will incrementally degrade product features when enabled, we go to great lengths to minimize its impact during testing.

Our strategy is to execute frequent, but *small scale* A/B tests to get continuous signals for Defcon knob resource savings as

well as potential issues, and infrequent large-scale exercises to validate these signals at scale and observe how the knobs for a product, service, or feature (and downstream services) behave when Defcon is enabled for all of the product's, service's, or feature's requests.

We classify production tests into two categories:

1. Small scale tests. A/B testing measuring user behavior metrics helps us quantify the impact on users and products. These tests are conducted across a product, but with a very small user base, (e.g., 0.01–2%) of a population over a certain duration (e.g., 15 minutes to 36 hours). The main goals from these tests are vaidating the knob set-up for the product, measuring the impact from the knobs, and measure the savings on downstream services using TRU. During A/B testing, we define four groups. One control group without any impact from Defcon and three treatment groups with different Defcon levels. We compare the resource consumption between the different groups with the control group as a baseline to measure the impact from Defcon on the service and the downstream services. To understand the impact of the Defcon knobs at a more granular level, we also run A/B tests at a service level (for any services that have knobs defined) and at a feature level.

We run server-based tests for multi-tenant backend services when per-user annotation is not propagated (e.g., for batch-processing services or multi-tenant services, where requests may belong to the system or several users simultaneously). In this case, we randomly select a small number of hosts for a particular service, and split these hosts in 4 groups, testing as described above.

We compare host metrics with the control group and store the results. The downside of this approach is that user experience may be momentarily inconsistent because consecutive requests from the same user may be served by different hosts. To minimise the user impact, we pick a negligible number of hosts for this test and run it for 5–15 minutes only. We run host-based tests weekly to always have fresh data and make sure that results are consistent. If results are not consistent, we adjust the number of randomly selected hosts. To make sure that our results are statistically significant and reliable we check that they match with empirical *large scale test* results (discussed next).

2. Large scale tests. Since Defcon is an emergency tool, we must test Defcon at scale to ensure its reliability and performance. We execute a Defcon service test on 100% of users quarterly to measure the resource savings at the product and service level and the demand reduction on the downstream services. Since during an emergency situation we may need support from Defcon for multiple products at the same time, we also execute combined degradation tests for multiple products together to measure the impact on Meta's infrastructure. During such tests, we enable Defcon knobs across products at levels 3, 2 and 1 for a short time and we monitor behavior similar to individual product tests.

3.5 Degradation Policy

Graceful feature degradation provides a trade-off between resource savings and product behavior. When designing the policy for when – and to what extent – to enable degradation, we must understand the trade-off between capacity savings from enabling a knob or collection of knobs and the user or product impact that comes from doing so. Product teams are responsible for defining key performance indicators that are closely measured and monitored during tests. Infrastructure teams provide a distributed tracing framework [19] to measure resource savings not only on the product, service, or feature where the knob is implemented, but also along the transitive closure of services affected by the knob.

Meta implements a four-level Defcon policy scheme whereby smaller-numbered levels correspond to *higher* amounts of degradation. Levels can be applied across the same features supported by the Knob Actuator Service (product, service, and feature). To ground the policy in reality, care has been taken to design each level around handling a specific set of failure scenarios:

Level 4 (L4) is the default state: All knobs are disabled.

Level 3 (L3) is used for handling overload situations resulting from relatively small-scale load spikes such as those seen during New Year's Eve or sporting events like World Cup. The Level 3 target savings is 5% of a product's overall demand.[3]

Level 2 (L2) is used for handling overload situation that arise from full data center region failures. Target savings is 10% of a product's overall demand (but can vary up or down based on a product's data center deployment model).

Level 1 (L1) is used during rare emergency events such as unforeseen global system outages. Target savings is 20% of a product's overall demand.

For setting target level savings, recall from Section 2.2 that Meta's infrastructure is organized around a collection of geographically-distributed data center failure domains, each representing around 5–12% of the overall capacity, making 5–20% of savings to be a sweet spot for mitigating the risk of cascading failures due to overload.

The Knob Definition Framework allows product developers the freedom and flexibility to explore potential knob resource savings and trade-offs in order to arrive at a portfolio of knobs that attempt to maximize the resource savings while minimizing the potential impact to users. When setting these level targets, service owners will translate demand reduction numbers to whichever resource they bottleneck on, like CPU,

[3] Most front-end services at Meta have CPU utilization as the bounding resource, and so target CPU savings is the most salient metric to focus on.

memory, network. We rely on historical data for the amount of demand increase typically seen during similar past scenarios.

Figure 4 illustrates the four-step policy that emergency responders follow when operating Defcon in production. Emergency responders:

1. *Analyze* the state of the product resource demands and potential Defcon knob resource supplies using a system monitoring dashboard. The dashboard lists critical system resource utilization metrics described in Degradation Policy.

2. *Decide* if the situation can be mitigated by applying a Degradation Policy. A degradation policy specifies the resource and impact trade-offs associated with enabling knobs at a particular level for a product.

3. *Check* the current state of Defcon and adjust it in accordance with the desired Degradation Policy (e.g., enable L2 knobs for a product).

4. *Degrade* fungible product features using a command line interface (CLI). Continue at step 1, adjusting Defcon level as necessary.

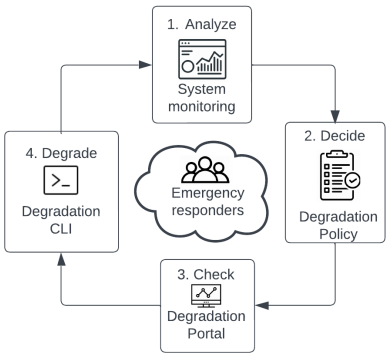

Figure 4: Emergency responders rely on a well-defined *Degradation Policy* in order to engage Defcon effectively.

We next evaluate the efficacy and trade-offs associated with operating *Gratuit* in a real-world large-scale environment.

4 Evaluation

At Meta, we have operated Defcon across three products – Facebook, Instagram, and Messenger – for over three years. Defcon has been used to avert or avoid many dozens of situations that would have otherwise led to resource exhaustion and overload. We next evaluate Defcon to demonstrate its efficacy, both during tests as well as during real-world events.

4.1 Measurement Methodology

We relied on four main sources of data for our analysis:[4]

1. A *Real-time Monitoring System (RMS)* for measuring hardware counter statistics across the entire fleet of servers at Meta to measure real-time demand for server resources.

2. A *Resource Utilization Metric (RUM)* source of truth data set for available server resource supply, measured using load-test data. Supply metrics include available request throughput, CPU MIPS, memory bandwidth, and so on.

3. A *Transitive Resource Utilization (TRU)* system that uses a distributed tracing framework to measure resource changes across the transitive closure of services involved in serving requests from a particular service.

4. A *User Behavior Measurement (UBM)* framework for quantifying any user workload changes that occur during a test.

Using these systems, we measure two *system-level* metrics during testing: (1) The global savings in resource utilization on the product, service, or feature under test using RMS; and (2) the savings in resource utilization on back-end services that receive traffic from the product, feature, or service under test using a combination of RMS, RUM, and TRU.[5]

We rely on controlled UBM experiments in order to measure the non–system-level effects of Defcon in a statistically significant manner. Requests to a product, service, or feature under test are divided into two groups: A control group (group A) and a test group (group B). Resource usage and user behavior is measured and then compared between group A and group B. Tests are run on a small fraction of users (typically a fraction of a percent) and over a long enough period to obtain statistically-significant results (typically minutes to hours).

In addition, for large-scale tests that involve large collections of knobs, we utilize various data science approaches to model each of our metrics both before a test (a forecast) and after a test (a backcast). Through linear modeling and time-series forecasting/backcasting, we construct a source-of-truth signal during the test period. Resource savings are subsequently computed by taking the percentage difference between the real signal captured during the large-scale test and the constructed source-of-truth.

As an example of this methodology, Figure 5 shows the measured global request throughput for the Facebook product before a product-level Defcon test. This experiment was performed as the product was nearing its peak moment of

[4]Due to space constraints, we do not detail the design of these systems in this paper.

[5]Whether to use RMS or RUM depends on the resource consumption to measure.

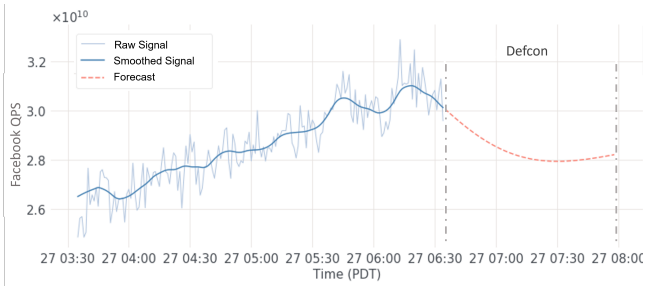

Figure 5: An example of timeseries forecasting. The measured global request throughput (QPS, y-axis) over time for the Facebook product immediately prior to enabling Defcon. A raw signal is converted to a smoothed signal and a forecast is generated from the smoothed signal.

request throughput (i.e., the highest organic load that we can test upon). Raw signal obtained prior to switching on Defcon ("raw signal") is first smoothed ("smoothed signal") and several time-series forecasting models are applied to obtain an estimate of what the raw signal would have looked like if Defcon was not turned on. Examples include linear, exponential, and Auto-Regressive Moving Average (ARMA) [28] models that are fitted using sections of the test signal before and after the Defcon degradation period.

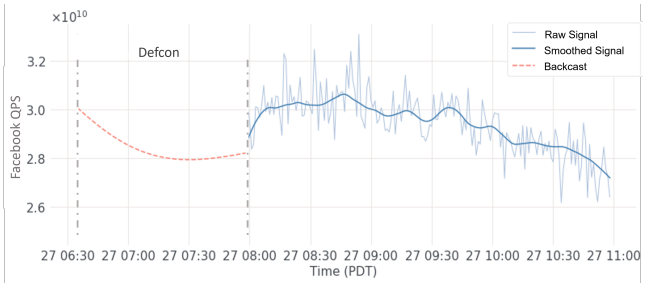

Figure 6: An example of timeseries backcasting. Methodology is similar to Figure 5.

Aside from the forecasting models, we also reference past days' signals in the steady state. The model which gives the smallest Median Percentage Error (MAPE) [23] is then chosen. Similarly, Figure 6 shows a time-series backcasting method applied to the smoothed signal gathered when Defcon is turned off. Note that the forecast and backcast use a somewhat conservative approach to ensure that measured savings are not over-estimated and to factor in headroom for the spikes observed in the raw signal. Finally, combining both the forecasted and backcasted signals (Figure 7), we derive a baseline which tells us what the metric would have been under normal circumstances when Defcon is not enabled. Savings are subsequently computed by taking the difference between the actual signal gathered during a Defcon test and the baseline.

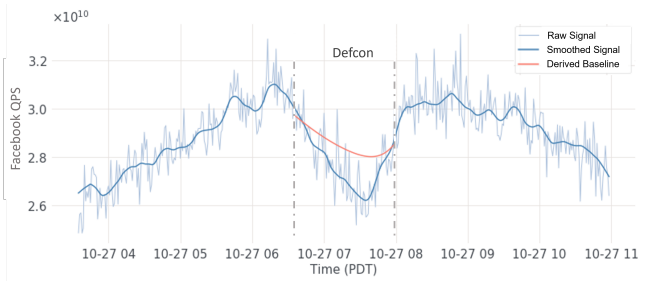

Figure 7: By combining forecasts and backcasts during a Defcon test, we can construct a baseline to compare to the behavior when Defcon is enabled during a test.

4.2 Individual Product Tests

To continuously validate Defcon savings and reliability, we regularly perform A/B tests with a small percentage of users (0.01%, 0.05% and 0.5% for Level 1, Level 2, and Level 3 experiment groups respectively). The user percentages are set based on required population size of A/B test statistical analysis. Figure 8 shows the results of A/B test applied across different product areas. The y-axis shows the CPU resource consumption (measured in relative MIPS) and each bar corresponds to a group under test. As we expect, enabling lower levels of knobs generally results in more resource savings.

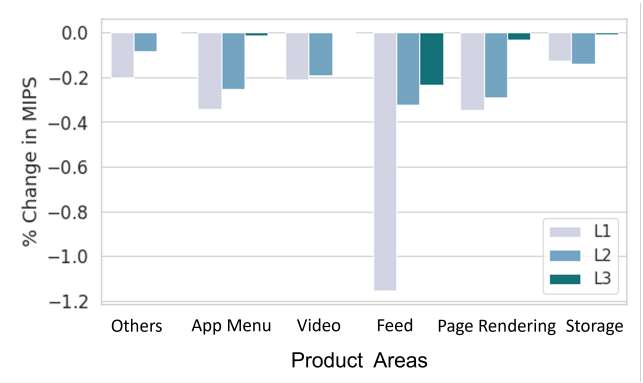

Figure 8: Results for Defcon tests for different sets of knobs (Product Areas) with different Defcon levels enabled. The y-axis plots the percentage change in CPU resource consumption, MIPS. Lower values indicate greater resource savings. Tests at each level are *inclusive* of *higher* levels (i.e., L1 tests also include L2 and L3 knobs). Notice that, generally, L1 knobs have larger resource savings than L2 knobs, and the same for L2 and L3. However, levels correspond to *impact* and not necessarily *savings*, and so some products, such as Storage, can achieve higher savings at lower levels of impact (L2 > L1).

Table 2 provides a detailed example of user impact metric data measured using the UBM framework described in Section 4.1 while testing at Level 1 for: (1) an individual knob, (2) a collection of knobs for a feature, and (3) all the knobs within all the features that make up a product. We observe

that degradation generally leads to relatively small user interaction changes, especially when compared to the alternative of an overload event leading to a site-wide outage. We also observe that enabling knobs can lead to user interaction *shifts* since user behavior changes in response to feature availability. For example, Video Watch Time at the Feature granularity increases when Level 1 knobs are turned on, as users engage in different ways to interact with a product.

Metric Name	Knob	Feature	Product
User Interaction	–1.82%	–4.3%	–5%
News Feed Usage	–0.6%	–1.1%	–1.6%
Video Watch Time	–0.6%	+2.37%	–0.93%
App Usage Time	–0.36%	–1.9%	–11.0%

Table 2: Example UBM metrics when enabling Defcon Level 1 for a selected Knob, Feature, and Product. User Interaction measures high level user engagement metrics for an app, like the number of comments, reactions, posts, and so on over the test interval. News Feed Usage is a composite metric measuring feed views and feed interaction time. Video Watch Time is a composite metric aggregating time spent watching videos, count of live viewers, engagement with live videos, and so on. The Knob granularity is for an individual knob defined for the product. The Feature granularity is the feature that contains that individual knob and all other knobs that make up the feature. The Product granularity is for the product that contains that feature and all other features that make up the product.

4.3 Combined Product Tests

At Meta, we regularly run combined degradation tests for multiple products. Figure 9 shows a combined Defcon test for three products: Facebook, a multi-tenant asynchronous compute platform (Async), and Instagram on 100% of traffic. The main goal of these tests is to accurately measure the combined transitive resource savings for shared backend services (here we illustrate the savings for Memcache, an in-memory key–value store [24]). As we can see, enabling Defcon across these three products leads to a compounding resource reduction for Memcache.

Of course, even when core product behavior remains unchanged, users may not expect to see changes in product features. At Meta, a user can submit a report when the user encounters something unexpected. Figure 10 shows user reports for four products during a combined test. As the figure shows, changing the features within products does not go unnoticed by users, with users submitting higher than nominal reports when Defcon is enabled. Note that this volume of reports – while keeping core product functionality available – is much preferred compared to fail-slow or overload conditions which could be several orders of magnitude larger without Defcon enabled.

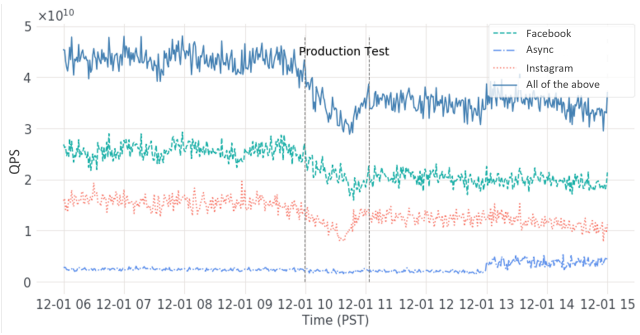

Figure 9: An example of transitive resource changes on a multi-tenant backend system (Memcache), measured in QPS (y-axis). Requests are tagged according to which source of traffic sent the request: Facebook, an asynchronous compute platform (Async) and Instagram. We see that Facebook and Async contribute the most to the reduction in overall QPS (All of the Above).

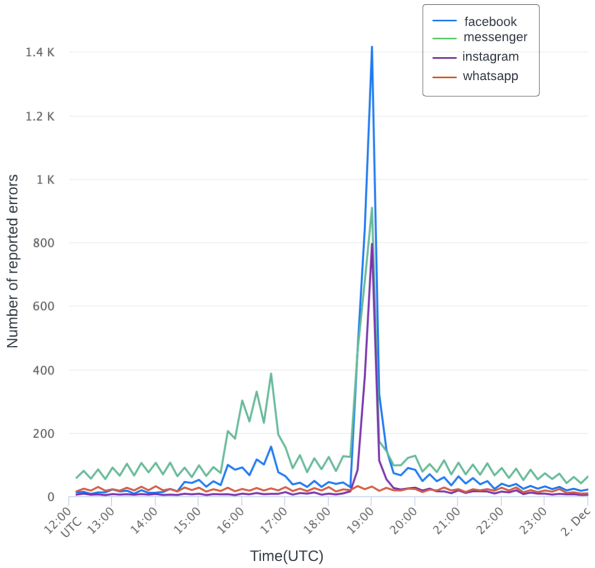

Figure 10: Number of reports submitted by users for different products during a combined product test.

4.4 Transitive Resource Savings

We next explore in more detail how transitive savings affect dependent services. Figure 11 shows an example of the resource savings achieved on the Memcache service (the same service from Figure 9) as measured *only* for the requests originating from the Feed product. Knobs of decreasing level were enabled incrementally during the test and then removed later in the test.

Note that knobs for the Feed product were only enabled during the first half of the annotated test range, and while other products participated in this test, using TRU, we were

able to observe the resource changes for only a single source of requests. Crucially, this savings is a beneficial *side-effect* of the reduction in workload from the front-end service and *not* a result of knobs defined in the Memcache service.

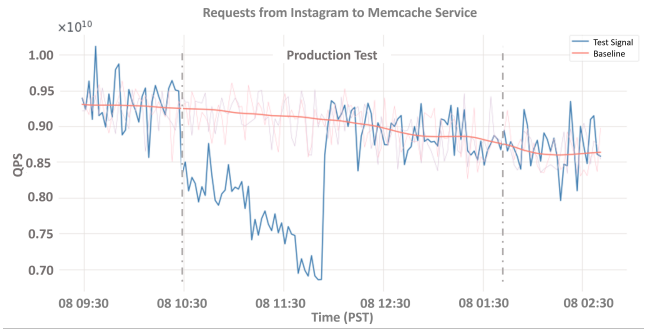

Figure 11: Resource savings, measured in QPS on the y-axis, on the Memcache service by enabling Defcon on upstream products, despite the Memcache service having no knobs defined. Reducing the request throughput to the Memcache service leads to corresponding reductions in resource consumption for the service *and its dependent services*.

Figure 12 shows an example of resource savings for TAO (a social graph caching service [7]) when Defcon is enabled across the three products under test. In addition to showing another service that achieves savings despite not having *any* knobs defined, it also shows an example of how resource savings can remain relatively stable over long periods of time (hours).

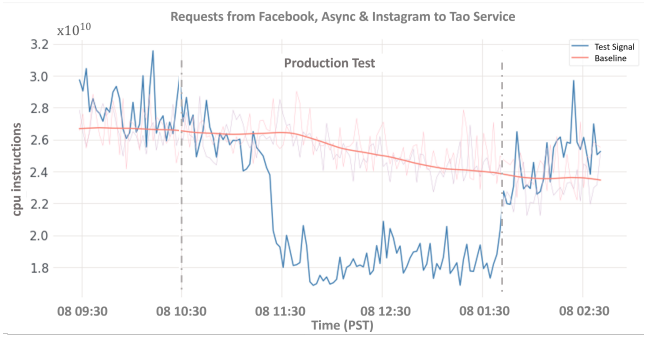

Figure 12: Another example of transitive resource savings on a social graph (TAO) service that has no knobs defined. We find resource savings from Defcon to be stable over long periods of time (e.g., hours).

Figure 13 adds annotations to the results for TAO, showing the distinct phases involved in a large-scale test. As we can see, products enable knobs of decreasing level until reaching Level 1, remain at Level 1 for a small period of time, and then return to a disabled state. In this case, we can clearly see in Phase III, IV, and V that most of the demand for the TAO service comes from the Async product. We record insights such as these as metadata for knobs and use the insights to inform decisions during real-world site events.

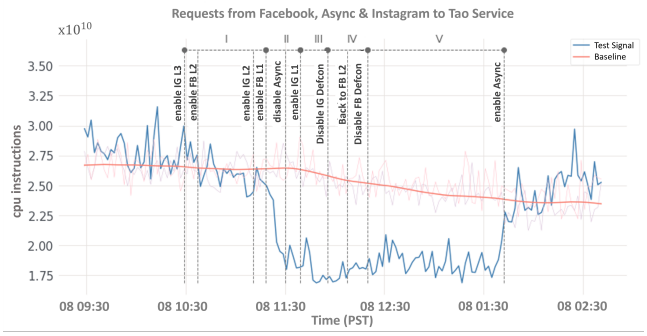

Figure 13: A detailed timeline of events during a typical multi-product Defcon test. This figure illustrates the complexity of testing Defcon at-scale in a production environment.

Interestingly, we also find that enabling Defcon across multiple products can achieve more resource savings for a product than enabling Defcon for that product alone. This occurs because some front-ends (such as the Facebook product) also serve RPC requests from other products (such as the Instagram product) so enabling Defcon on the other products reduces the resource consumption of the Facebook product. Figure 14 shows such an interaction for the Facebook product during a test when Defcon is applied to the asynchronous compute product, Async. We can see that even after Facebook knobs are disabled (around 17:50 UTC), Facebook still sees reduced resource consumption compared to its baseline due to reduced requests from the Async product.

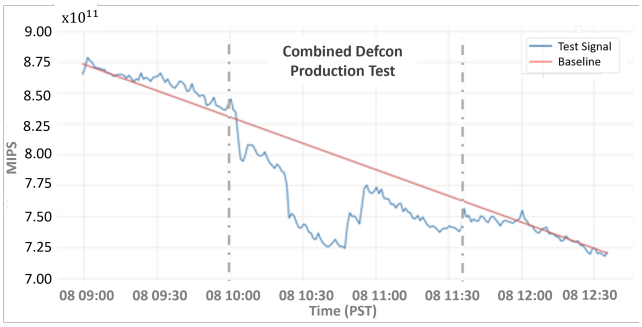

Figure 14: An illustration of the inter-dependent resource savings of knobs: Enabling knobs for the asynchronous compute product, which sends requests to the Facebook product, leads to additive savings compared to enabling knobs for the Facebook product alone.

4.5 Outage Simulation Testing

At Meta we also simulate the conditions posed by large-scale outages such as natural disasters by redirecting traffic away from data center regions in order to concentrate more traffic

on the remaining regions, akin to what could happen during a fiber cut, hurricane, or power grid failure [33]. This reduces the available resource supply, effectively simulating load spike events such as New Year's Eve or World Cup.

In Figure 15 we show the results of running a large-scale test on the Facebook product through the Facebook product's peak moment of traffic. At the beginning of the test, we sequentially redirect traffic from multiple data center regions (labeled C and A) in order to concentrate enough load on the remaining regions. This operation continued until site operators began to detect a small volume (measured in very low parts per million of requests) of failed requests due to overload, whereupon Defcon was enabled at Level 2.

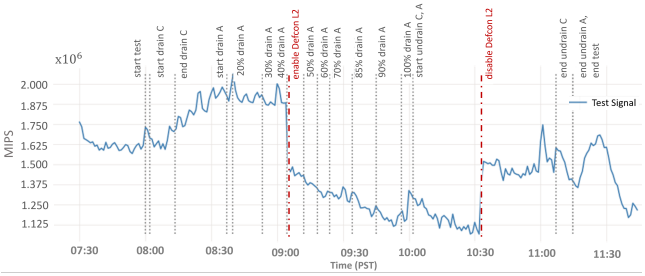

Figure 15: We regularly run tests to test the efficacy of Defcon under large-scale outages. In this test, we start by redirecting product traffic away from two data center regions in order to reduce the amount of server resource supply for the same amount of user demand, thereby increasing the resource utilization of the remainder of the fleet. We then enable Defcon in order to validate resource savings when products are in a highly-loaded state.

Moreover, after enabling Defcon at Level 2, we *continued to redirect traffic* until the second data center region was completely drained of traffic. This example illustrates how Defcon can effectively avert overload conditions that could ultimately lead to fail-slow behavior and wide-spread cascading failures. Tests such as this also provide valuable validation of the measured resource savings in a realistic environment: At-scale, at peak, and using the real production workload.

To illustrate the importance of at-scale testing, in Figure 16, we show an example of measured resource savings on a separate day, during a similar time, using the same knobs as the previous example. We can see that while the mean resource savings during this test is similar to the real-world increased load simulation, it is not exactly the same. A major reason for this is cold cache effects from traffic being redirected among data centers, a realistic concern during real-world outages.

To illustrate the generality of our approach, Figures 17–20 show similar results across four different products – Feed, TAO, Memcache, and Graph Search – during a different two–data-center region drain test at peak levels of traffic with L2 knobs enabled. The y-axes of these figures have been normalized to compare the relative sensitivity to knobs across different products, with the measured savings corresponding

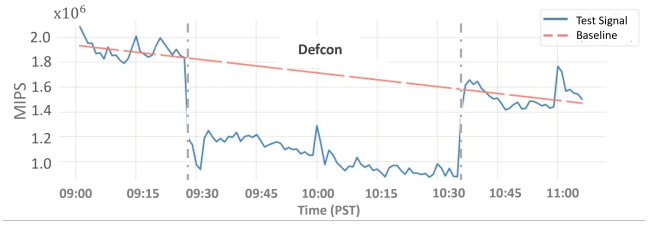

Figure 16: We find that resource savings (measured in MIPS on the y-axis) are *load-dependent*. In this example, having warm caches *increases* the amount of resource savings (corresponding to lower values of MIPS) compared to when outages are simulated (cf. Figure 15).

to 3.2% for Feed, 2% for TAO, 8% for Memcache, and 6% for Graph Search.

Notice that while different products achieve different levels of savings (these are reflections of their own target savings for L2 knobs), their response to enabling Defcon can vary due to caching effects and workload pattern changes in response to enabling knobs. The figures also illustrate how different products can customize their demand metrics used to measure and track their target Defcon savings (e.g., by using CPU Cycles or Power consumption).

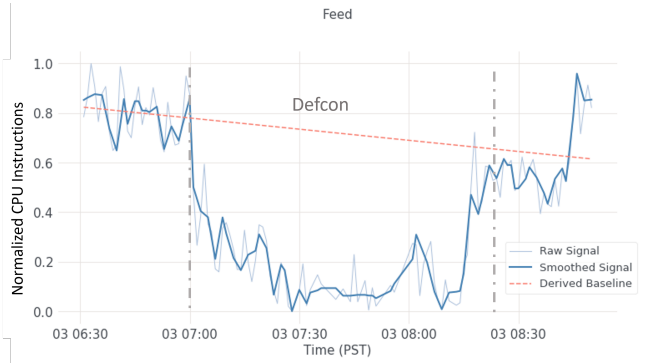

Figure 17: Results for L2 knobs enabled during a two–data-center region drain test for the Feed product.

4.6 Real-World Large-Scale Outage

Since Defcon is an emergency tool used during large-scale outages, we must ensure that unknown unknowns are minimized. Based on the different degradation tests that we execute for products, and by measuring the impact on users and downstream services, we work closely with Site Reliability Engineers (SREs) to come up with degradation policies and guidelines for the scenarios where Defcon can help. During a real-world outage, SREs work with a lead emergency responder, the *Incident Manager (IM)*, who decides on which options from the Degradation Policy to pursue to mitigate an outage.

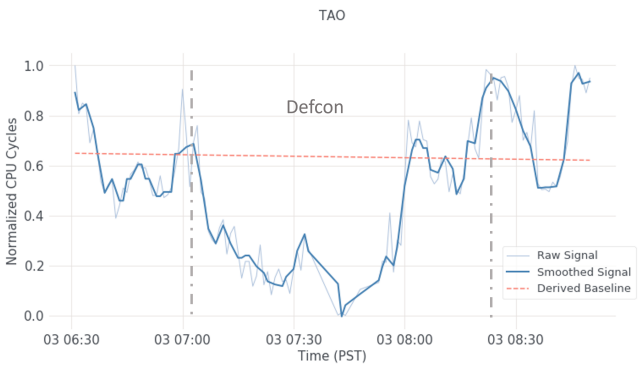

Figure 18: Results for L2 knobs enabled during a two–data-center region drain test for the TAO product.

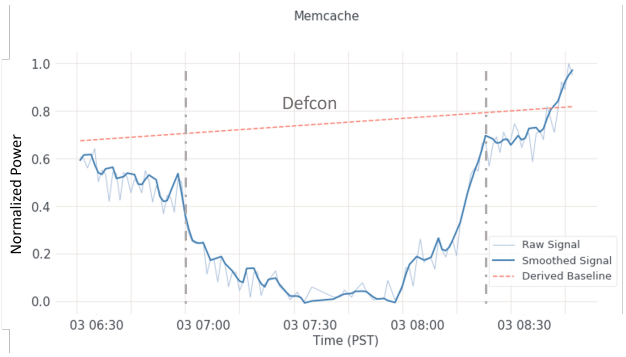

Figure 19: Results for L2 knobs enabled during a two–data-center region drain test for the Memcache product.

Figure 1 from Section 1, showed one such outage where the IM applied the principles listed in the Degradation Policy to avert a site-wide overload event and outage (please refer to Section 1 for an detailed explanation). During this event, the IM made the call to first engage L3 knobs for the product before eventually engaging L2 knobs. The fact that different levels of knobs – with different amounts of impact – existed, provided the IM with a spectrum of options to pursue in order to eventually arrive at the right degradation trade-off in real-time.

To ensure that we could mitigate this real-world event smoothly, we needed to ensure that the degradation policy discussed in Section 3.5 has been practiced by SREs and the IM. To make sure all the responding members are trained on using the policy, we frequently execute mock fire drills where we come up with potential scenarios, and role play the necessary steps to mitigate the risks. We have found such testing to be largely beneficial in ensuring emergency responders are prepared when disasters strike.

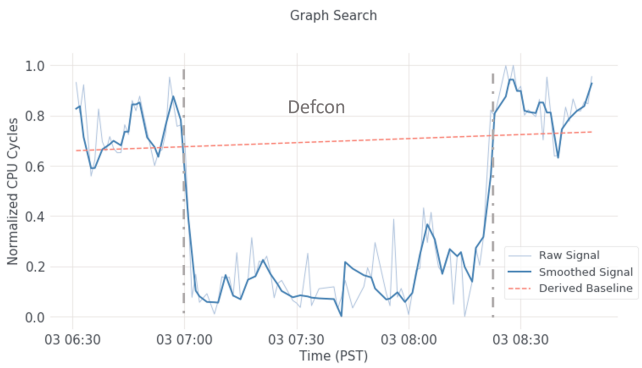

Figure 20: Results for L2 knobs enabled during a two–data-center region drain test for the Graph Search product.

5 Lessons Learned

Over the past several years of using Defcon, we have learned several key lessons to consider for graceful feature degradation:

1. Understand business goals and customer perception to determine what to degrade.

Prior to implementing knobs, product engineers first decide on which features to degrade. Core product functionality must remain intact, but among the non-core features, we find that there exists a spectrum of resource savings compared to user impact. For this reason, product designers perform A/B tests (cf. Table 2) and make a decision about which knobs to keep and which to pass on. While this process requires human interaction, the Knob Definition Framework and Knob Testing Framework allow developers to quickly explore the knob definition space in order to determine the set of knobs that provide the most resource savings for the least user impact.

2. Leveraging graceful degradation during emergencies requires regular testing and an easy-to-consume understanding of the business and customer impact.

To provide an easy-to-consume understanding for emergency responders to use in the heat of the moment, product engineers provide a high-level *functional* summary of what is affected at each Defcon level. Using this summary, site incident managers can quickly determine whether enabling knobs for a product at a given level is an adequate response. Additionally, this summary benefits the public relations and communications team, who may need to respond to inquiries from customers or the media about product feature changes.

3. Degradation systems require high and regular commitment from product teams.

To motivate product engineers to work on Defcon knobs, we built mechanisms to provide recognition for investing in this technique for product reliability. We organize monthly Defcon meetings per product to showcase each team's work to their organizational leader (e.g., a vice president). We also

leverage the concept of Defcon champions. A Defcon champion is someone who is passionate about reliability that can drive Defcon throughout the organization. Defcon champions identify and recruit people in their organization to work on Defcon.

4. Knobs, once built, need to be regularly maintained.

Implementing and maintaining knobs requires engineering effort. Identifying candidate features involves coordination with product developers to run experiments to understand the capacity savings and user impact of knobs. Developers, however, have provided feedback that controlling and testing knobs using a standardized framework has helped them to rapidly develop and deploy knobs. While automated systems measure and report knob behavior, regressions in capacity savings and user impact require manual investigation. We intend to explore automating this area of knob maintenance in future work.

5. Low dependence and high availability actuation.

To ensure that Defcon is ready to be deployed during disasters, we iterated on improving our operations and operational availability. As an example, we developed a CLI with minimal dependencies on other systems in our infrastructure to make sure that Defcon is ready to be enabled during partial failures and disasters. Having a low dependence and highly available mechanism for knob actuation is critical for facing real-world disasters.

6. Developer experience and efficiency are key.

Before Defcon existed, there were scattered independent efforts to try to achieve similar goals. By unifying these disparate efforts and providing tools to support teams in a structured manner, we were able to increase the coverage of Defcon and simplify knob maintenance. Since Defcon is built on top of existing tools at Meta, such as Configerator [32], developers do not need to learn new technologies to implement new Defcon knobs.

Safety is handled by ensuring that features are isolated at the RPC layer (a design practice at Meta) and thus knobs typically encapsulate control flow between RPC callers and callees. While fine-grained degradation within a binary serving an RPC request is possible, safety and consistency must be validated by product developers during initial knob testing. We note that such validation is similar to what developers must do when routinely modifying binary control flow (i.e., not for the purpose of Defcon knobs) – a common practice at Meta. To aid developers in knob definition, we provide guidance on how to properly implement and maintain knobs, as well as provide developers with a Knob Testing Framework to measure Defcon savings and track regressions.

The main challenge for developers in maintaining Defcon knobs is capacity savings regression tracking. Systems at Meta are constantly evolving, so the impact of existing knobs can drift over time. Because of this, we make sure that each team tests Defcon savings at a limited scale in production at least once every three months (an interval chosen to balance knob impact with the need to understand behavior changes) using the Knob Testing Framework. We are actively exploring ways to test knobs more frequently at lower impact.

6 Conclusion

We presented Defcon, a system for graceful feature degradation to prevent overload in large-scale Internet services. We hope that by characterizing the overload problem, the corresponding solution space, and our approach to graceful feature degradation, we will spark discussion within the research community about how best to tolerate overload-induced system behavior and advance reliable and available distributed system design.

References

[1] Trustworthy Graceful Degradation: Fault Tolerance across Service Boundaries. https://www.usenix.org/conference/srecon21/presentation/rodgers-prior, 2021.

[2] Tarek F. Abdelzaher and Nina Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31(11–16):1563–1577, may 1999.

[3] Satyajeet Singh Ahuja et al. Network entitlement: contract-based network sharing with agility and SLO guarantees. In *SIGCOMM'22*, 2022.

[4] S. Almukhaizim, T. Verdel, and Y. Makris. Cost-effective graceful degradation in speculative processor subsystems: the branch prediction case. In *Proceedings 21st International Conference on Computer Design*, pages 194–197, 2003.

[5] Matteo Maria Aurizzi, Tommaso Rossi, Emanuele Raso, Ludovico Funari, and Ernestina Cianca. An SDN-Based Traffic Handover Control Procedure and SGD Management Logic for EHF Satellite Networks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 196(C), sep 2021.

[6] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable Failures in Distributed Systems. HotOS '21, page 221–227, 2021.

[7] Nathan Bronson, Zachary Amsden, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. USENIX ATC, 2013.

[8] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301, 2012.

[9] Google Cloud. Infrastructure Design for Availability and Resilience. https://services.google.com/fh/files/misc/infrastructure_design_for_availability_and_resilience_wp.pdf, 2020.

[10] Shuai Ding, Sreenivas Gollapudi, Samuel Ieong, Krishnaram Kenthapadi, and Alexandros Ntoulas. Indexing Strategies for Graceful Degradation of Search Quality. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, page 575–584, 2011.

[11] Google Site Reliability Engineering. Addressing Cascading Failures: Load Shedding and Graceful Degradation. https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degredation, 2019.

[12] Haryadi S. Gunawi et al. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *FAST'18*, 2018.

[13] M.P. Herlihy and J.M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[14] Hideaki Hibino, Kenichi Kourai, and Shigeru. Difference of Degradation Schemes among Operating Systems — Experimental analysis for web application servers —. In *Proceedings of DSN 2005 Workshop on Dependable Software - Tools and Methods*, pages 172–179, 2005.

[15] David Ke Hong, Qi Alfred Chen, and Z. Morley Mao. An Initial Investigation of Protocol Customization. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '17, 2017.

[16] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.

[17] Lin Huang, I-Hong Hou, Sachin Sapatnekar, and Jiang Hu. Graceful Degradation of Low-Criticality Tasks in Multiprocessor Dual-Criticality Systems. pages 159–169, 10 2018.

[18] Xiaofan Jiang, Jay Taneja, Jorge Ortiz, Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, David Culler, Philip Levis, and Scott Shenker. An Architecture for Energy Management in Wireless Sensor Networks. *SIGBED Rev.*, 4(3), jul 2007.

[19] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. SOSP, 2017.

[20] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building More Robust Cloud Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 700–711, 2014.

[21] HyunJong Lee, Shadi Noghabi, Brian Noble, Matthew Furlong, and Landon P. Cox. BumbleBee: Application-Aware Adaptation for Edge-Cloud Orchestration. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 2022.

[22] Jingqiang Lin, Bo Luo, Jiwu Jing, and Xiaokun Zhang. GRADE: Graceful Degradation in Byzantine Quorum Systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 171–180, 2012.

[23] Spyros Makridakis. Accuracy measures: theoretical and practical concerns. International Journal of Forecasting, 1993.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry Li, Ryan McElroy, Michael Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkat Venkataramani. Scaling memcache at facebook. NSDI, 2013.

[25] Alessandro Vittorio Papadopoulos, Jakub Krzywda, Erik Elmroth, and Martina Maggio. Power-Aware Cloud Brownout: Response Time and Power Consumption Control. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017.

[26] Jeremy Philippe, Noel De Palma, Fabienne Boyer, and et Olivier Gruber. Self-adaptation of Service Level in Distributed Systems. *Software: Practice and Experience*, 40(3):259–283, 2010.

[27] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, Availability, and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service. *ACM Trans. Comput. Syst.*, 18(3):298, aug 2000.

[28] Björn Schelter, M. Winterhalder, and J. Timmer. *Handbook of Time Series Analysis: Introduction and Overview*, chapter 1, pages 1–4. 2006.

[29] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. Incentivizing Self-Capping to Increase Cloud Utilization. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 52–65, 2017.

[30] Mohammad Shahrad and David Wentzlaff. Availability Knob: Flexible User-Defined Availability in the Cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 42–56, 2016.

[31] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *ACM Trans. Storage*, 1(2):133–170, may 2005.

[32] Chunqiang Tang, Thawan Kooburat, Pradeep Venkat, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. SOSP, 2015.

[33] Kaushik Veeraraghavan, Justin Meza, Sankaralingam Panneerselvam Scott Michelson, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. OSDI, 2018.

[34] Jagannadh Vempati, Ram Dantu, Syed Badruddoja, and Mark Thompson. Adaptive and Predictive SDN Control During DDoS Attacks. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–6, 2020.

[35] Julien Verlaguet and Alok Menghrajani. Hack: a new programming language for hhvm. https://engineering.fb.com/2014/03/20/developer-tools/hack-a-new-programming-language-for-hhvm/, 2014.

[36] J. Robert von Behren, Eric A. Brewer, Nikita Borisov, Michael Chen, Matt Welsh, Josh MacDonald, Jeremy Lau, Steve Gribble, and David Culler. Ninja: A Framework for Network Services. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, June 2002.

[37] Eugene Wiehahn and John Walker. Target Group Load Shedding for Application Load Balancer. https://aws.amazon.com/blogs/networking-and-content-delivery/target-group-load-shedding-for-application-load-balancer, 2021.

[38] David Yanacek. Using load shedding to avoid overload. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload, 2020.

[39] Lidong Zhou, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Roy Levin, and Chandramohan A. Thekkath. Graceful Degradation via Versions: Specifications and Implementations. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 264–273, 2007.