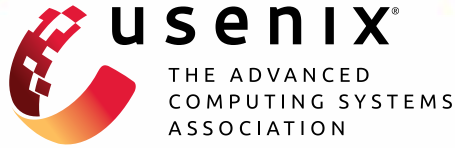

Global Capacity Management With Flux

Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang, *Meta*

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

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

Global Capacity Management With Flux

Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang

Meta Platforms

Abstract

Customers of both private and public clouds must wrestle with the problem of *regionalization*: how should service capacity be apportioned across a large number of geo-distributed datacenter regions? This problem is further complicated by the complex service dependency graphs that arise from microservice architectures, as well as capacity availability and hardware mix that can vary greatly by region.

Historically, regionalization has been solved through a slow-moving and manual process, whereby owners of large services directly negotiate capacity allocation and distribution with the cloud provider. However, as both service and cloud footprints continue to grow, these manual processes are becoming untenable, and often result in excessive labor for all parties involved, as well as suboptimal outcomes.

At Meta, we have built a system called *Flux* to automate capacity regionalization, transitioning it from a bottoms-up, manual process, to a top-down, automated one. Flux employs RPC tracing to identify service capacity models, and uses these to compute an optimal joint capacity and traffic distribution plan that spans thousands of services across tens of products, and involves millions of servers. These plans are orchestrated by a system that safely and efficiently rebalances service capacity and product traffic across tens of regions on a continuous basis.

1 Introduction

Meta's private cloud consists of millions of machines and hosts products serving billions of users. It must provide the products with a growing, geographically distributed capacity footprint, so that they can scale and remain fault tolerant while their usage grows and new features are introduced.

Our products employ large microservice architectures [29] comprising thousands of services, globally deployed in tens of datacenter regions. Most of these services are *shared* by multiple products, and hence the sizing of one service needs to consider the demands of all products. Moreover, the services are *interdependent*. It is not uncommon that a service calls tens of other services and the depth of the call graph can go beyond 10 levels. As a result, capacity distributions for services must be managed in concert: the growth of one service may cause the growth of tens more, which in turn places further capacity demands on their downstream services, and so on. Thus, it is a daunting task to manage capacity at scale, as service operators must answer questions such as: How to size my service? How and where do I provision capacity for organic growth, or to enable a new feature? Are downstream services correctly provisioned for the demand generated by my service?

To avoid the above complexities getting out of control when planning capacity jointly across tens of regions, service owners often prefer the simplicity of reasoning about their capacity on a per-region basis: a service responds to demand increases by requesting that new capacity be delivered to the regions where the service is already running.

However, this local optimization leads to many issues:

- Services in mature regions cannot grow as those regions have no available space or power to add capacity.
- Hardware utilization becomes imbalanced as some regions may provide more capacity than others.
- Hardware ordering is overly constrained by specific services requiring specific hardware to be placed in specific regions.
- Capacity imbalances lead to excess disaster-readiness buffers as we must have enough buffers for the potential loss of the single *largest* region.

Before Flux, these issues were solved by lengthy negotiations between service owners and cloud providers. For example, in order for service A to grow in region X, the service owners might negotiate to trade the service's excess capacity in region Y for service B's capacity in region X. This laborious process does not scale well with the number of regions and services, and leads to suboptimal capacity allocation. Moreover, even after capacity negotiations, rebalancing services and traffic across regions in order to match capacity supply is still a daunting task, requiring coordination across many services and traffic distribution systems.

Finally, this region-centric capacity management process leads to tight coupling between specific products and specific regions. The *capacity mix* (i.e., the ratio of different hardware types) of any given region often reflects the products that have historically been deployed to that region. As a result, the capacity mixes of our regions have already diverged significantly, further exacerbating the problem as this regional heterogeneity limits the flexibility of moving services across regions to rebalance demands and supplies.

Global capacity management. Our key insight to solving these issues is to elevate capacity management to a *global* problem, decoupling global service placement from global hardware placement. This paper describes our global capacity placement system, called Flux, which runs continuously on weekslong timescales, redistributing service capacity and product traffic to best utilize our global capacity footprint.

Flux utilizes RPC tracing to identify a model that predicts service capacity demands given a traffic mix for our products. This model is then used to formulate a mixed-integer-programming (MIP) problem that jointly distributes service capacity and product traffic across all of Meta's regions. Flux's service placement distributes service growth capacity, rebalances existing service capacity to meet infrastructure goals, and manages projected regional capacity deficits such as those caused by regional hardware refresh.

A *capacity orchestrator* works with our existing autoscaling and traffic management systems to safely and efficiently execute global capacity redistribution plans. The orchestrator also allows human-in-the-loop operations as needed by escalating low-quality estimates to be vetted by human operators, or delegate orchestration for services that have not yet onboarded to our autoscaling systems.

Flux has been running in production at Meta for 2.5 years, continually allocating service capacity quotas and performing cross-region shifts of service placements and traffic for our largest products. Currently, Flux covers about 50% of the servers in our private cloud that consists of millions of servers supporting online, batch, and AI training & inference workloads. We expect Flux to cover more than 90% of our capacity as we increase adoption. Flux has also enabled dramatic simplification of our hardware planning process, as it allows us to plan for capacity globally, while gradually homogenizing our current heterogenous regional hardware mixes.

Besides Flux's usage in our private cloud, the ideas presented in this paper can potentially be adapted to public cloud settings as well. Public cloud providers also similarly negotiate with their large customers directly to match capacity demands with supplies. This sometimes entails providing capacity outside of the customer's preferred regions, which in turn may require customers to relocate their workloads.

Contributions. We make the following contributions:

- To the best of our knowledge, this paper is the first to conduct a comprehensive study of global capacity management and global service placement, which are important issues for public and private cloud providers. We hope that by sharing our firsthand experiences, we can help the research community better understand this important problem and the constraints involved in solving it.
- We propose *global* capacity contracts, whereby service owners only need to reason about their global capacity demands, leaving it for Flux to optimally *regionalize* service capacity and product traffic distributions. By contrast, cloud providers still mostly operate in a mode where large services require specific hardware to be placed in specific regions, and traffic distribution is not integrated with capacity management.
- We use RPC tracing to build a service-capacity regionalization model, which calculates a service's regional capacity distribution as a function of the traffic mix for different products. We are not aware of any prior work that attempts to use models to regionalize service capacity. Moreover, although RPC tracing has been used for debugging and performance modeling, we are not aware of any prior work that uses it for capacity modeling, not to mention doing it at our scale and in production.
- We formulate a MIP problem to optimally distribute service capacity under constraints of capacity supply as well as service and infrastructure objectives. Despite the widespread use of MIP, our approach is novel in its application to joint capacity and traffic regionalization, a problem that has not been considered before. We also use load-test-induced nonlinear models to complement MIP-based linear models, improving modeling accuracy.
- We describe our *capacity orchestrator* which integrates across autoscaling and traffic management systems to safely implement capacity and traffic redistribution plans. Automating joint service placement and traffic redistribution at our scale is highly risky and may negatively impact site reliability. To our knowledge, this has not been attempted before.
- Finally, the effectiveness and robustness of Flux is demonstrated by the fact that we use it every quarter to assign hundreds of thousands of new machines to services.

2 Background

In this section, we provide background on our datacenters, workloads, capacity management practices, and the capacity management challenges that are addressed by Flux.

2.1 Datacenter Regions

Meta operates 10s of datacenter regions, each comprising multiple datacenter buildings in the same local geography, typically located on a single campus.

Our existing infrastructure abstractions generally operate at the level of regions. For example, our cluster management

systems [37, 45] manage all machines in a region as a single pool. Services are expected to be oblivious to the placement of their tasks within a region, and they may be spread across multiple datacenter, network segments, or power domains. Our regional infrastructure abstractions are supported by a network fabric that provides sufficient regional cross-sectional bandwidth for all but a few workloads

2.2 Traffic Management

We maintain a global network of small edge datacenters that are connected to our backbone network. User traffic (e.g., those from apps or web browsers) are terminated in these edge datacenters. Requests from clients connected to our edge datacenters are forwarded to front-end servers in one of our geo-distributed regions. A traffic distribution system [18] manages the distribution of requests from edge datacenters to our large datacenter regions, typically by considering a combination of factors including front-end utilization and geographic proximity to the end-user.

2.3 Service Workloads

Traffic enters a region through a front-end web service, typically an HTTP reverse proxy that routes the request to an appropriate application server based on the request URI. The application server implements some business logic, and typically makes RPC calls to tens to hundreds of services, which in turn fan out to yet more services. While some of these services implement functionality used by just a single product, most are shared across many products. Thus, our services are highly interdependent, and we cannot partition our services into product-specific silos.

Figure 1 illustrates the complexity in our request serving paths, with a large fanout and deep call depth. The complexity of service interdependencies [29] motivated us to use precise RPC tracing to attribute resource consumption when designing Flux, rather than using indirect methods such as statistical analysis [2] or heuristics [3, 38, 43].

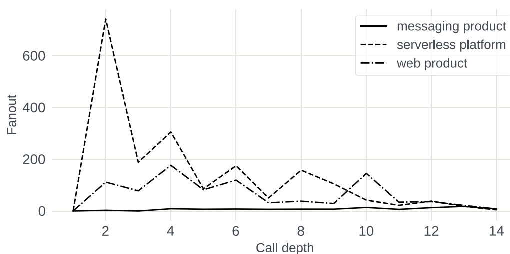

Figure 1: **Service RPC fanout.** An example of how to read the curves: when a request for the `web product` reaches the call depth of 10, it fans out to call 158 services on average.

2.4 Service Capacity Management

Meta's capacity management systems provide quotas in the form of regional reservations [37], which provide strong guarantees of regional capacity availability and sub-regional failure tolerance. Thus, the various hardware buffers required to reliably operate services within a region are encapsulated by the regional capacity management systems, and hidden from higher-level capacity management systems like Flux. The number of service replicas in a region is usually determined by our autoscaling systems which combine service capacity models with demand forecasts and disaster scenario simulations to ensure the job is sized correctly.

Most RPCs occur within the same region due to strict latency requirements imposed by applications. Additionally, our complex service dependency graphs often contain critical paths with tens of hops, which can quickly amplify cross-region RPC latencies. Finally, by hosting both the caller and callee in the same region, we can limit cross-region dependencies and improve disaster readiness [31, 52].

Figure 2 illustrates regional caller-callee affinity by showing the latency distribution of RPC calls across thousands of compute services, denominated by total capacity. The inflection point at around 25ms represents calls going cross-region; observe that $\approx 80\%$ of capacity is reached by in-region requests.

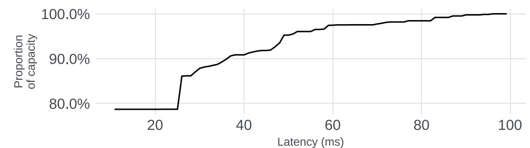

Figure 2: **Cumulative distribution of capacity by RPC latency.** Note the y-axis begins at 80%.

All of our products are located in multiple regions to improve disaster readiness, access a large capacity pool, and achieve wide geographic distribution. However, the distribution of service capacity across regions is uneven. This is due to the organic growth of both service capacity demand and regional capacity supply. Service capacity demand responds to new features and world events, while regional capacity supply depends on factors such as power availability and datacenter construction timelines. Additionally, geographic skew in product usage exacerbates the issue as we try to place service capacity close to end users.

Over time, this has led to a negative feedback loop, wherein services prefer to grow proportionally to their existing regional footprints, causing regional hardware mixes to reflect these historical workloads. This in turn makes it difficult to move these workloads to other regions, causing services to continue preferring the regions in which they are already deployed to receive capacity growth. We can see this specialization reflected in the regional hardware mix, as shown

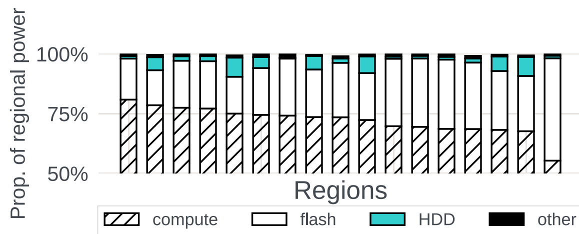

Figure 3: **Regional hardware type distribution.** A histogram of the compute, flash, HDD, and other hardware deployed across a subset of our regions. Note that the y axis starts at 50%. The right-most region is dominated by flash, because it is a small region that recently underwent a large retrofit, leaving a lot of database workloads in place.

in Figure 3. For instance, the percentage of servers of the `compute` type ranges from 55% to 80%.

Regions also vary in their power headroom, i.e., the difference between used and available power. Observe in Figure 4 that six regions have little available power, implying that if service deployments in those regions need to grow, they must expand their capacity footprint in other regions as there is little additional power to support additional racks. This is similar to the situation in public clouds where users cannot acquire new capacity in a given region [10–12, 21, 22, 47].

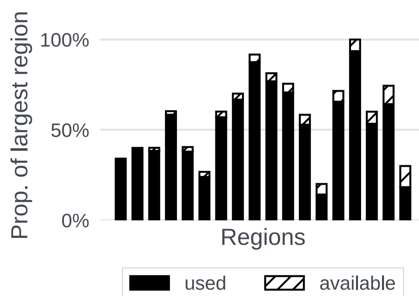

Figure 4: **Power headroom per region for capacity growth.** The y-axis is power, normalized to our largest region.

2.5 Capacity Management Challenges

Service capacity management presents significant challenges for both service owners and infrastructure operators. While service owners wish to grow freely where they already are deployed, infrastructure operators must reconcile these wishes with constraints associated with operating physical infrastructure, as well as goals around efficient fleet operations.

These problems are amplified as regions reach maturity–when there is no longer additional power available to allocate to new racks–and infrastructure owners cannot accommodate service capacity growth without shrinking the footprint of other services. The complex dependency graphs between services in a typical online product make this a more challenging problem still.

We refer to this challenge as the *service regionalization problem*: How can we optimally allocate capacity to a set of services across multiple regions? Additionally, how should product traffic be appropriately distributed based on this allocation? Finally, considering that cluster and capacity management systems usually operate at the regional level, how can we effectively rebalance services according to the regionalization plan?

3 Design and Implementation

Our global capacity management system, *Flux*, continually rebalances a large number of interdependent services across regions in response to demand changes (e.g., product growth) and supply changes (e.g., hardware refreshes). By decoupling the management of capacity demand and supply, Flux enables service owners to focus on their global capacity demand without considering regional needs, and allows cloud providers to evolve each region's capacity supply independently.

3.1 Overview of Flux's Workflow

As illustrated in Figure 5, Flux solves the regionalization problem through the following workflow.

Product-to-service capacity attribution via RPC tracing. Flux uses RPC request tracing [30, 40] to construct a regional *baseline* ① that attributes each service's peak capacity footprint to the products that are served directly or indirectly by the service. This baseline is used to construct a service placement model ② that determines how service capacity should be distributed given a product traffic distribution.

Joint regionalization of service capacity and product traffic. Using inputs from our budgeting systems, Flux then creates a capacity *target* for each service, which specifies the amount of global capacity that is needed for each service. These service targets are jointly regionalized with product traffic by formulating an assignment problem that is solved using mixed-integer programming (MIP) ③. The result of this stage is a *placement plan* that redistributes service capacity and product traffic across regions. The optimization problem also encodes a number of infrastructure objectives, such as minimizing the total amount of disaster-readiness buffer required to operate services safely.

Global capacity orchestration. Flux introduces a global capacity orchestrator, responsible for executing placement plans safely and efficiently ④. The orchestrator drives automation through several capacity and traffic management systems, and also supports human-in-the-loop operations to handle exceptions or uncertainties in execution.

The overall Flux workflow runs continuously in weekslong cycles to rebalance service capacity across regions according to changing hardware supplies and service demands. It measures the current state of service and hardware placement, computes a desired state, and then executes a plan to move

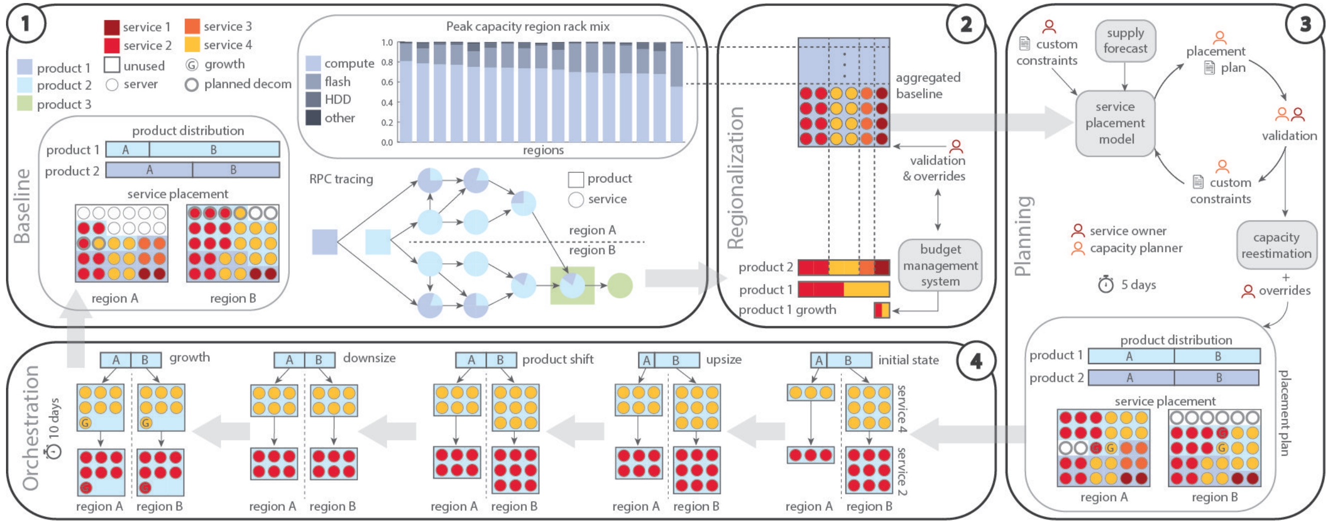

Figure 5: **Flux's end-to-end workflow.**

towards the desired state. It tolerates imperfect execution of the plan by repeating this self-correcting reconciliation loop.

Next, we describe the details of the steps above.

3.2 Service Modeling

The goal of service capacity modeling is to determine the optimal distribution of a service's capacity across regions in concert with product traffic. This is challenging because many of our services are shared by multiple products, and each product may invoke different call paths within the service. This can result in significant differences in the cost per request depending on the product being served (see §2.4).

3.2.1 Baseline

Flux defines a *baseline* for each region, attributing portions of each service's peak capacity footprint to different products. This baseline is created by combining two other baselines:

Capacity usage baseline. We run profilers on every server in our fleet to produce a dataset that attributes resource usage to specific services. Profiles are sampled every minute, and we process this dataset to identify the daily peak time window[1] and peak resource usage per service and per region, covering different resource types such as CPU and SSD.

Demand baseline. Flux uses sampled RPC traces to reconstruct the call graphs for requests that are handled by each service. We identify a set of *product gateways* that act as traffic entry points for each product. Importantly, the traffic destined for these gateways is globally and independently routable, and is usually managed by Meta's shared traffic management systems [18]. Each sampled RPC call is attributed

[1]Demand spikes due to new product launches or special events such as New Year's Eve are handled separately. During daily off-peak periods, many of our services are automatically scaled down to donate unused capacity to our elastic capacity pools, which are used to run preemptible services.

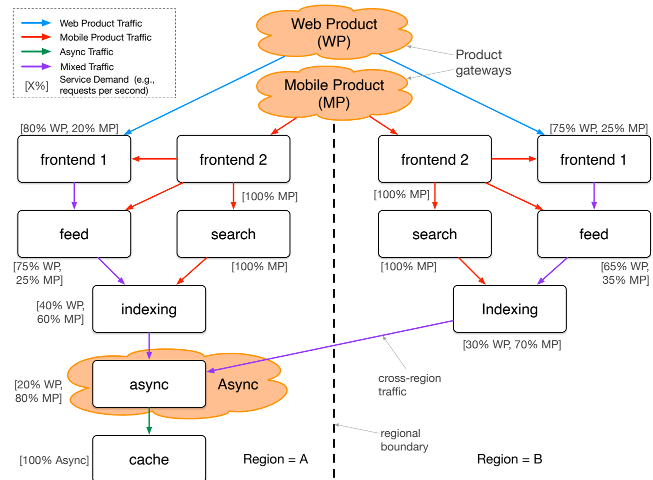

Figure 6: **Demand attribution.** Attribution of capacity usage to three Products: `Web Product`, `Mobile Product`, and `Async`. For the `indexing` service in `Region A`, the annotation "`[40% WP, 60% MP]`" means that 40% and 60% of the indexing service's capacity consumption is attributed to the `Web Product` and `Mobile Product`, respectively. Note that while demand attribution is relative, the capacity usage baseline is defined in terms of absolute capacity.

to the product handled by the nearest upstream gateway in the call path. The demand attribution process is illustrated in Figure 6. Sampled traces are aggregated to compose the demand attribution dataset for each service, dividing a service's total demand among the set of products served by it. Traces collect multiple demand metrics, including call counts as well as CPU instructions. The next section discusses how we select the demand metric that minimizes the overall model error.

3.2.2 Modeling

Flux's service models predict the amount of service capacity required in a region to serve that region's product traffic mix. We assume that each product's traffic is fungible across regions, and thus that capacity requirements for serving a fixed portion of the product's traffic is also the same across regions.

This suggests a model where capacity for service s in region r, $c_{s,r}$, is given by a linear combination of the capacity contributions from each product:

$$c_{s,r} = \sum_{p \in P} (\alpha_{s,p} * \rho_{p,r}) + \delta_{s,r}, \quad (1)$$

where $\alpha_{s,p}$ is the amount of capacity for service s attributed to product p; $\rho_{p,r}$ is the *presence*, or the proportion of global traffic for product p assigned to region r; and $\delta_{s,r}$ is the model residual for service s in region r.

Modeling residuals may be due to an existing capacity imbalance, or due to nonlinear effects not captured by the model. We allow service owners to be involved in managing the treatment of residuals during planning.

The baseline includes multiple demand metrics. In the modeling step, we select the metric that minimizes modeling error. For example, many large services have per-request costs that vary significantly across different products (because they invoke different internal code paths), and are well-modeled using CPU instructions as a demand metric. On the other hand, some services perform very little computation for each request. Therefore, call counts are a more appropriate demand metric for these services due to the CPU overhead of tracing.

3.3 Joint Capacity & Traffic Regionalization

Flux computes joint service capacity and traffic regionalization plans by formulating an assignment problem that is solved by a MIP solver. This section provides the intuition behind the problem formulation.

The formulation, detailed in appendix A, is an optimization problem that jointly assigns capacity for each service in each region and product traffic in each region, corresponding to $c_{s,r}$ and $\rho_{p,r}$ from equation 1. The assignments are subject to a set of constraints imputed from the service placement model described in §3.2, which determines the service capacity mix required to serve each product.

Initial capacity and traffic assignment are given by the baseline. The capacity residual ($\delta_{s,r}$ in equation 1) is interpreted as capacity that is unexplained by the model, and is thus excluded from reassignment, unless directed otherwise by the service owner. We provide an analysis of capacity residuals in §6.3.

Baseline adjustments. Flux adjusts the existing *baselines* to match planned capacity and product distribution changes. These planned changes are encoded as events and maintained in a *capacity ledger*. Flux commits its plans to the ledger along with other capacity planning systems. Thus, Flux can overlap planning and execution, as we can adjust the baseline to account for planned changes between the plan generation time and the start of its execution cycle.

Regional capacity pools. Flux divides each region's capacity into a shared pool per hardware type. Our capacity reservation system, RAS [37], provides these pools as a regional abstraction that we build upon, and lets us treat the capacity in each pool fungibly.

Each hardware type is assigned a capacity measure that represents the common bottleneck for that hardware type. For example, the generic compute pool is denominated by a normalized CPU throughput measure, while our SSD hardware is generally I/O bound. The capacity measure is normalized across all generations of the same hardware type. Some services can run on multiple hardware types: we encode this knowledge through a set of fungibility rules that establish a service's conversation ratios between different hardware types.

Service capacity demand. The *global* capacity demand for each service is computed by querying our budgeting systems which mandate service capacity budgets in terms of a normalized cost measure. Flux converts this normalized budget to a hardware-type specific capacity demand using a conversion ratio specific to the service and hardware type.

Service placement model. Flux imputes placement constraints from the service capacity model. Specifically, for each service, the model determines a *lower bound* of service capacity assigned to a region as a function of the product traffic mix to that region (see §3.2.2). The product traffic assignments are also optimization variables, and hence the service and traffic placement is jointly optimized. The baseline model residual (see Equation 1) is codified as an explicit term in the formulation to offset capacity imbalances that exist at baseline. Flux gives service owners the choice to reduce the model residual, which is often used to correct baseline capacity imbalances; see §6.5 for an example.

Optimization constraints. The MIP assignment problem constrains (1) the capacity assignment in each region to be no more than its available supply; and (2) the global capacity demand for each service to be met. The latter constraint is a soft constraint, which allows us to prioritize capacity fulfillment among services if necessary. Flux prioritizes *baseline* capacity footprint (i.e., the capacity present when the baseline was measured) over *growth* capacity (i.e., additional capacity granted by the budget systems), to ensure that already granted service capacity is not taken away.

Optimization objectives. The MIP assignment problem includes several infrastructure objectives that help us manage our global capacity footprint more effectively. First, a balancing objective spreads service capacity evenly across regions, which reduces the amount of buffer capacity needed

for disaster readiness. Second, unused capacity is also distributed evenly based on region size, making extra capacity available to account for discrepancies or defects in Flux's placement. Third, a stability objective minimizes the amount of capacity reassignment in each placement cycle, simplifying the placement plans and reducing infrastructure churn.

Timesteps. Regionalization is simultaneously computed for multiple future timesteps in increments of future execution cycles. This serves three purposes. First, multi-timestep plans can incorporate large changes in supply or demand ahead of time, allowing for a plan that anticipates these changes with small, individually feasible adjustments over multiple timesteps. Second, we can set stability objectives that span multiple timesteps to prevent undue oscillation in placement plans. Finally, this multi-timestep approach prevents the solver from optimizing a short-term solution at the expense of long-term negative impacts.

While Flux computes plans for multiple timesteps, we only execute the plan for the next timestep. Flux runs in a self-correcting reconciliation loop as the baseline can change between executions for a number of reasons, including: (1) execution may have deviated from the placement plan; (2) supply and demand forecasts may change in the interim; (3) manual capacity operations may affect the baseline; (4) service code changes or new services may affect the service models.

The formulation is detailed in appendix A.

3.4 Execution Planning

Flux derives an *execution plan* from the joint service and traffic placement plan. The plan is a directed acyclic graph (DAG) of service capacity assignments and product traffic assignments. The plan ensures that services are always sufficiently provisioned for the traffic during the transition stage.

Flux provides this guarantee through a three-phase plan. First, all upsizes are executed, i.e., each service is sized to the maximum of its baseline size and its target placement size. Second, all product traffic is reassigned. Third, downsizes are executed by sizing each service to its target size.

The advantage of this approach is its simplicity and the ability to execute it quickly by parallelizing actions in each phase. A disadvantage is that it requires temporarily overprovisioning services, which takes up capacity that could be used by other services. To address this limitation, we explored using more sophisticated multi-step plans. However, we found that these plans were both complex to execute and difficult to explain to service owners. As a result, we decided to continue using the simple approach.

3.5 Orchestration

The execution plan is fulfilled by Flux's capacity orchestrator which: (1) executes capacity and traffic assignments in the correct dependency order; (2) continuously monitors product-level and service-level metrics to ensure that they remain healthy; (3) delegates exceptions and actions to human operators as needed; and finally (4) performs load tests to validate the placement.

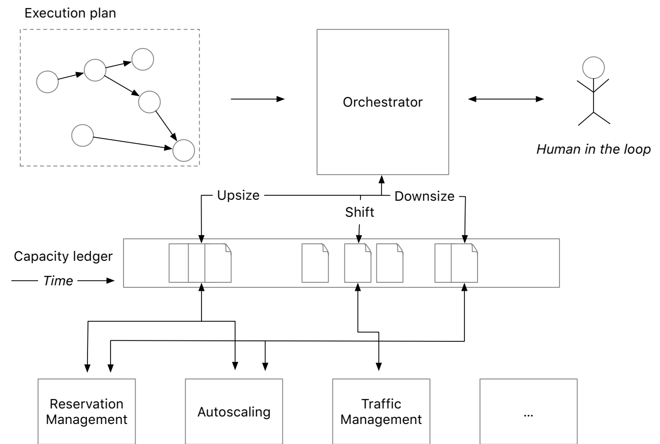

Figure 7: **End-to-end orchestration workflow.**

Figure 7 illustrates the orchestration workflow. A *capacity ledger* stores a timeline of *capacity events*.These events are timestamped, and each reflect a proposed capacity related change. Capacity and traffic management systems query the ledger for future events, but only execute them once they are marked by the orchestrator as *active*. After execution, the same systems store their status (success or failure) back into the ledger.

The ledger acts as a central repository of all anticipated capacity changes, and allows multiple systems to simultaneously propose and coordinate changes, while decoupling *capacity planning systems* from *capacity management systems*. While Flux is the primary writer to the ledger, we sometimes write manual events to make temporary capacity changes in support of product launches or experimentation.

The ledger provides three important properties. First, we can compose events from different writers, so that the underlying management systems can consider the combined effect of a set of events. Second, by providing events ahead of time, we accommodate services that require a long lead time to provision capacity and scale. For example, our caching systems need a significant warm-up period before newly provisioned capacity can handle production traffic. By providing future events, the control plane gives the management systems enough time to ensure that services are ready for future traffic shifts. Third, the ledger serves as an authoritative forecast of future capacity changes, and is used by Flux to incorporate future and ongoing events during planning. This allows Flux to overlap planning with execution, and to compose well with other capacity planning systems.

The orchestrator ensures that events are executed in dependency order by verifying that all antecedent events have

completed before marking an event as active. RF delegates any exceptions to human operators through its UI. The UI is also used when (1) the service owner has configured the service to require validation before execution, or (2) the capacity estimates for a service are considered low confidence in the modeling stage and require operator validation. By providing this progressive path towards full automation, Flux offers transparency and explainability, and allows service owners to gain comfort with the system.

Finally, before downsizing service capacity, the orchestrator initiates product load tests [52] to validate that the sizing is correct and that the site as a whole remains disaster ready.

3.6 Stateful Services

Flux integrates with Shard Manager [32] to handle stateful services within our platform. Shard Manager is responsible for managing most of our stateful workloads. Shard Manager continuously queries the capacity ledger for relevant capacity events, and builds new replicas after upsize capacity has been provided by Flux. This is done by migrating or replicating data from other regions. Shard Manager then acknowledges the capacity event, allowing Flux to safely proceed with execution. After the demand shift, Shard Manager safely removes the old replicas from downsized regions before Flux reclaims capacity.

The primary challenge with integrating stateful services today is that the default demand attribution algorithms do not always accurately capture requests costs. Such systems often exhibit interaction between requests, where processing of one request can affect the cost of subsequent requests. Our default attribution algorithms also do not capture persistent storage costs, the effects of caching, etc. We work with service teams to update our algorithms to better capture their capacity cost models. For example, TAO [16], Meta's social graph store, maintains a custom cost model, which captures many of the above effects across their complex, distributed system. We integrate this cost model into Flux's attribution models to correctly capture TAO's capacity needs.

4 Design Alternatives

In this section, we discuss the major design alternatives.

4.1 RPC tracing

Flux relies on RPC tracing for gray box measurements of product-service capacity attribution. Meta has invested in a unified RPC stack [39], leading to high out-of-the-box tracing coverage without any additional instrumentation needed from service owners. Moreover, all our main traffic ingestion systems [30,52] already implement sampled trace origination.

As of 2022, we have 52% of capacity usage covered by RPC tracing. For services that are not yet covered by RPC tracing, we have been working closely with the the service owners to drive the adoption, because distributed tracing [40] as a fundamental capability in a large infrastructure has broad usage beyond capacity management, such as problem determination [17] and performance debugging [2].

Black box methods like statistical analysis [2] or heuristics [3,38,43] can be used to infer service call graphs without needing service-specific instrumentation. However, our highly interdependent microservice architecture makes employing such techniques less accurate. Since many of our backend systems are shared among multiple frontends, which invoke distinctly different callpaths, often with substantially different cost per request.

Black box methods were previously only evaluated on simple three-tier applications, while in our complex environment, the depth of call graphs reaches 14 and the RPC fanout is as high as 742, and hundreds of different upstream services may call a given service at varying call-graph depths. The full complexities of Meta's service topology and call graphs are reported in detail in a recent work [29]. These complexities make statistical or heuristic methods less applicable to our environment.

Furthermore, because we can mandate high tracing coverage in our services, we can expect higher quality models, which in turn helps us provide greater levels of automation in global capacity management.

4.2 Nonlinear Service Models

The core service model used by Flux is linear: it assumes that capacity usage is linearly related to a chosen demand metric (see Equation 1) and a product traffic mix. While such models are simple to identify and to apply broadly, many services exhibit nonlinear capacity behaviors. When available, Flux can update its estimates by using more accurate nonlinear models such as those produced by load testing [55], queuing analysis [41], or by simulation.

Many of our services use continuous load testing to maintain an accurate model of the relationship between a service's capacity usage and its RPC throughput. These models are used by our *Capacity Estimator* (CE) [14] to ensure that services are sized correctly for demand and remain disaster ready as determined by simulating various failure scenarios. However, simulations introduce nonlinearities that cannot be represented in a MIP assignment problem. To solve this problem, Flux invokes CE with the traffic distribution produced by the MIP solver. CE then runs its simulations against the proposed traffic distribution, and provides updated capacity estimates that incorporate planned failure scenarios.

We have found that using a combination of a default linear model for regionalization, along with a load-test-induced nonlinear model for improving estimates, works well in practice. Linear models provide upper bounds and are amenable to MIP optimization, while the nonlinear models provide higher accuracy, and usually operate within the bounds of the linear

models. Additionally, the simpler linear models are easier to explain and diagnose. Currently, 9.2% of services using Flux have load-test-induced models, and these services account for 46.4% of the total capacity allocated by Flux. This suggests that larger services are more concerned about capacity and are more likely to build their load-test-induced nonlinear models.

5 Discussion

In this section, we discuss the challenges of doing capacity planning in a complex real world and several ways of applying ideas in Flux to public clouds.

5.1 Practical Challenges

Flawed baselines impact modeling accuracy. Because Flux started with a baseline that was the result of many years worth of ad-hoc capacity management, the baseline itself does not reflect an ideal placement. Thus, when modeling capacity, we have to take extra care when interpreting model residuals: they could be due to imperfect modeling, or baseline itself not well-balanced. Flux provides *rebalancing* to service owners wishing to correct these imbalances in a controlled manner.

Complete capacity models are hard to come by. We have found that many capacity management practices rely on tribal knowledge, ad-hoc modeling, and implicit agreements between services. These are not captured in the service capacity models that Flux operates with, and thus cannot incorporate various de facto objectives or constraints. We work with service owners to codify these, but often find that we need to work around these with manual planning adjustments. We have also introduced tools like the capacity ledger (see §3.5) that help capture and mechanize previously ad-hoc capacity management practices.

These realities mean that there isn't a clear ground truth for capacity distribution, and require a nuanced interpretation of both modeling residuals and execution errors.

5.2 Applying Flux in Public Clouds

Many large public cloud customers maintain "virtual private clouds", whereby they acquire large capacity pools of reserved instances. For example, Netflix has reported [36] that it runs thousands of services on hundreds of thousands of geo-distributed reserved instances [9] in AWS. These customers can apply Flux to manage these pools of reserved instances, and intelligently place services so that they are maximally utilized. These customers can also use Flux to extract recommendations about the type, location, and amount of capacity to acquire in order to accommodate growth, and to optimize the customer's capacity footprint.

Cloud providers could provide a new kind of capacity contract, whereby customers are guaranteed low-cost capacity, but are not guaranteed specific regional placement. The cloud provider offers an online capacity-planning tool through which their customers continually update their aggregated placement constraints. The cloud provider then regularly re-regionalizes the capacity for customers that use this form of capacity, calling into customer's control planes to execute service and traffic rebalancing. This is similar to existing preemption APIs for spot instances [8].

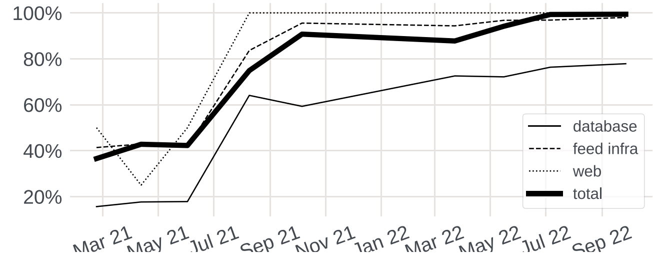

Figure 8: **Automated actions in total and by services.** Actions are nodes in the Flux's capacity placement plan, and include capacity and traffic management operation, such as adjusting regional quotas or resizing services.

6 Evaluation

Our evaluation answers the following questions:

1. How long does it take for Flux to execute its plan (6.1)?
2. Do Flux's service models help accurately assign global workloads to hardware in individual regions (6.2, 6.3)?
3. To what extent does Flux help meet the growing needs of out-of-region hardware refresh (6.4)?
4. How does Flux plan capacity and service placement for a specific service in practice (6.5)?

6.1 Execution Automation

Our goal for Flux is to maximize automation across both planning and execution, while incorporating humans-in-the-loop to review proposed actions and catch defects. We have granted Flux increasing autonomy as we gain confidence in the completeness and accuracy of Flux's models, its solvers, and automated execution systems. Currently, not all services support automation when adjusting their deployments across regions; Flux compensates by incorporating human-in-the-loop manual actions.

Figure 8 plots the degree of automation in Flux's execution plans, showing a handful of service groups as well as the overall automation. The drastic improvement in "*total*" around July 2021 was due to the introduction of a fully automated capacity distribution mechanism that integrates with our autoscaling system [14]. Over the past 2.5 years, we have increased automation in Flux from 27% to nearly 100%.

Figure 9 shows Flux's plan completion times. When Flux was first introduced, execution was dominated by operations requiring human feedback or execution. As we have simultaneously improved automation coverage and model quality,

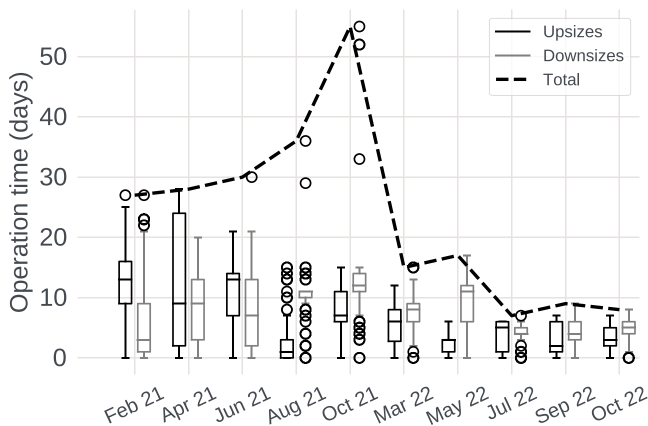

Figure 9: **Plan completion times.** The "*total*" curve represents the end-to-end plan execution time. The boxplots represent the distribution of individual service resize operation's completion time. From bottom to top, the markers on a box represent, excluding outliers, the minimum, lower quartile, median, upper quartile, and maximum. Outliers are shown separately as small circles. The spike in the "total" curve represents a large shift in the complexity of Flux's placement plans, which caused service automation coverage to lag Flux's capacity coverage. We spent the ensuing months improving the coverage of our capacity automation tooling.

fewer operations require human-in-the-loop intervention and scrutiny, and most operations are now fully automated. Recently, executions take roughly 1 week. Even with model and automation improvements, some limits still remain. For example, cross-region data replication or cache warmup may still require long execution times for stateful services even if they support full automation.

6.2 Capacity Sizing Error

We define Flux's capacity sizing *error* as the service capacity eventually used in production, minus Flux's recommended capacity assignments, which include improvements from using load-test-induced models when available (§4.2). The errors exist for several reasons. First, since capacity-planning mistakes can be costly, service owners often review and sometimes revise Flux's recommendation based on their domain knowledge of their services. Second, after Flux executes its capacity plan, our autoscaling system [14] may resize services in production, if it finds that additional capacity is needed to support Flux's traffic shift, or that a service is left with a capacity surfeit.

Figure 10 shows the proportion of upsize and downsize capacity executed in each plan. A value of 100% means that execution was (in aggregate) exactly to plan. Over the course of the last year, we have improved planning accuracy significantly, primarily by working with service owners to improve their attribution and capacity models. We use execution history to incorporate expected error rates into Flux's planning assumptions, and thus are able to tolerate this error by ensuring that we both (1) have sufficient capacity to support anticipated (aggregate) upsizes; and (2) are able to reclaim sufficient capacity where this is needed for refresh.

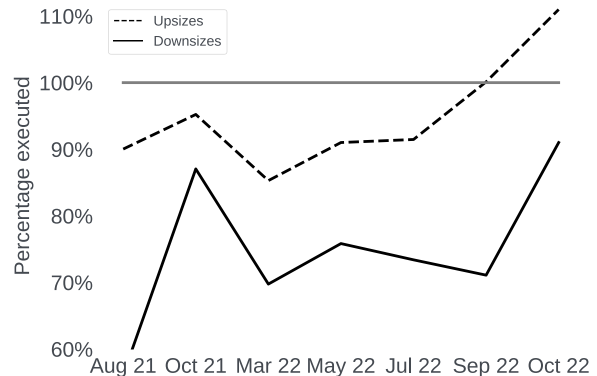

Figure 10: **Capacity-sizing error,** the percentage of Flux's recommended assignments executed in production. Each data point represents the proportion of the total capacity in a Flux placement plan that was actually executed. The error is split by upsizes and downsizes.

Unless service owners explicitly opt out, we require them to review Flux's placement plans through a UI tool. An Oct 2022 execution plan had 377 resize nodes. The total number of nodes available for Flux to execute on is about 3000. Of these, 81 nodes were for services that opted out of review; of the remaining 296 nodes, 84, or 22% of the total nodes, were revised by service owners. These revisions modify the resize node directly; Flux continues execution with the updated node. Our tool captures the reason given by service owners for each override, and we present the most frequent reasons below, with the number of each kind of override shown in parentheses.

Insignificant capacity (15). The service owner rejected the plan as the service does not have fully automated capacity management and the plan moved an insignificant amount of capacity. Therefore, the overhead to the service owner is too high to justify the benefits of execution.

Service should not be rebalanced (13). The service owner rejected the plan because the service should not be rebalanced by Flux. The remedy is to add the service to Flux's execution blocklist.

Insufficient headroom (10). The plan would leave a service without enough headroom capacity, usually to accommodate anticipated growth. The remedy for this is to capture this requirement as a capacity event, so that it can be incorporated into Flux's capacity plans.

Deprecated service (8) The service is deprecated, and should no longer be managed by Flux.

Bad estimates(4) The service owner judges the estimates to be incorrect, usually due to one of two reasons. First,

Flux has incorrectly attributed product demand to the service. In these cases, we repair the demand baselines, for example, by choosing another demand metric. Second, the linear model is inaccurate for the service. In these cases, we work with the service owner to adopt the load-test-induced model (see §4.2).

These overrides highlight the complexity of operating in a large-scale production environment. These results show that, with supervision, it is feasible for Flux to operate in a complex environment. Over time, as bugs are fixed and new features (e.g., headroom modeling) are added to Flux to cover a broader set of scenarios, we expect Flux to perform with higher accuracy and gain greater autonomy.

6.3 Model Residuals

The model *residual* δ (expression 1 in §3.2.2) measures the portion of baseline service capacity distribution not explained by Flux's service model. While the previously presented *error* metric reflects plan defects that could cause inefficient placement or operational risks, δ merely reflects the imbalance between traffic and capacity distribution. Large residuals often reflect pre-existing capacity imbalance or inherent limitations in services which prevent the system from achieving an ideal balance.

Figure 11 shows the residual for several representative services. `Web product`'s residual varies between 3% and 5%. Such stateless services are usually well-modeled by Flux.

The residual for `feed infra` was initially higher at ≈8%, because its capacity distribution was uneven before Flux was applied. Over multiple placement cycles, we have used Flux to reduce this capacity imbalance, which is reflected in recent residuals that match those of `web product`. Due to limited capacity supply during COVID, Flux's optimization objective has been dominated by supply constraints, and once Flux is able to meet decommission and growth objectives, we limit the infrastructure changes that Flux is allowed to introduce. As capacity supply improves in the future, we plan to use Flux to more aggressively rebalance service capacity and reduce the residual. The short-term variance of the residual curves in Figure 11 correspond to load tests [6] and drain tests [52], as well as Flux traffic shifts. These events temporarily distort the relationship between traffic and capacity distribution, and are filtered out of Flux's baseline.

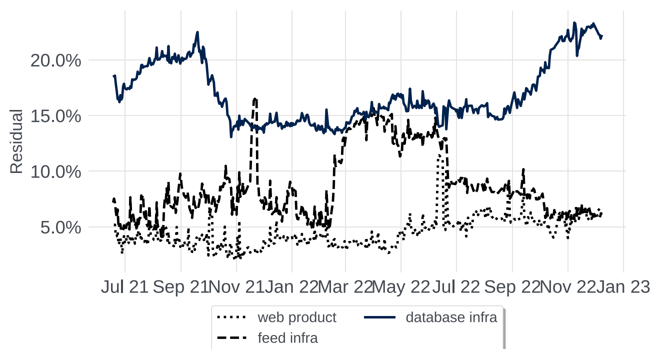

Figure 11: **Aggregate model residual.** We show $\frac{\sum_{r\in R}|\delta_{s,r}|}{\sum_{r\in R} c_{s,r}}$, for several representative services.

Figure 12 shows the distribution of `feed infra`'s model residual across regions. This kind of plot guides us to work with service owners to improve their service balance by applying more aggressive balance objectives in Flux. The figure also demonstrates that, while the aggregate residual is higher at ≈5%, the per-region residual is generally less than 1%.

In Figure 11, `database infra`'s model residual is the highest due to some unique challenges associated with stateful services. For example, if a subset of hot data shards are the bottleneck, naively adding more capacity may not improve the service's throughput proportionally. Many stateful services also have substantial capacity requirements for internal data replication [5], which fall outside of the usual request-response RPC regime, making it difficult to apply RPC tracing. We have been continuously improving Flux's support for stateful services. In Figure 11, the initial reduction of residual from 22% to 15% was primarily due to improved attribution accuracy and coverage for `database`. The later regression coincides with deployment of new `database` systems into just a subset of regions, and for which we need to define new product attribution rules to capture correctly.

Overall, we deploy many stateful services, including databases, storage, and caches. Of the capacity currently managed by Flux, 14% is for stateful services. Our long-term strategy is to migrate stateful services to a common stateful framework called Shard Manager [32], which solves many common problems (e.g., hot shards) that impact stateful-service modeling. Moreover, Shard Manager is integrated with Flux so that the services it manages are automatically covered by Flux. Finally, Shard Manager intelligently places data shards onto the capacity allocated by Flux to minimize data-access latency [1,4].

6.4 Accelerating Out-of-Region Refresh

Figure 4 shows that as regions mature, they may have minimal power headroom to accommodate new hardware. Accordingly, the "*quarterly OORR demand*" data points in Figure 13 shows the rapidly increasing need for performing out-

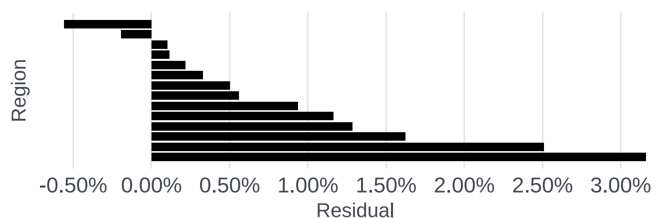

Figure 12: **Model residual for `feed infra` by region.**

of-region hardware refresh (OORR). Prior to 2020, a negligible amount of OORR was performed, and even then OORR was already a significant planning challenge. This was due to the lack of planning tools to compute global workload shuffling and traffic shifting for interdependent services, as well as lack of automation to execute such plans even when it was manually built. The benefits of Flux go far beyond OORR, but the imminent increase in OORR demand and the unsustainable toil in performing OORR motivated us to develop Flux. Flux has helped scale OORR planning and execution by ≈950% year-over-year. Currently, we perform global workload shifts once every 6 weeks; each shift typically reshuffling capacity for 100k-300k servers globally. The shifted capacity exceeds OORR demand because (1) the decommissioned capacity may not reflect the overall workload hardware composition, meaning that Flux must perform larger reshuffles to utilize the underlying hardware; and (2) Flux also allocates growth capacity and optimizes other infrastructure goals such as reducing disaster-readiness buffers.

6.5 Case Study: `FeatureStore`

As a detailed case study, we present the impact of Flux on `FeatureStore`, a flash-based key-value store serving features of machine learning models, deployed across thousands of servers, with a 99th percentile read latency of under 15ms, and a read request rate of 10s of millions per second. In September 2020, Flux was used for the first time to generate a service-placement plan for `FeatureStore`. Prior to that, its service placement was performed by humans.

A key event that took place during the September 2020 planning cycle was large-scale decommissioning of servers in three regions A, B, and C, resulting in a reduction of supply in those regions. Accordingly, we expect Flux to perform out-of-region hardware refresh and shift traffic away from those regions to others in order to accommodate this regional supply reduction, which is confirmed by Figure 14.

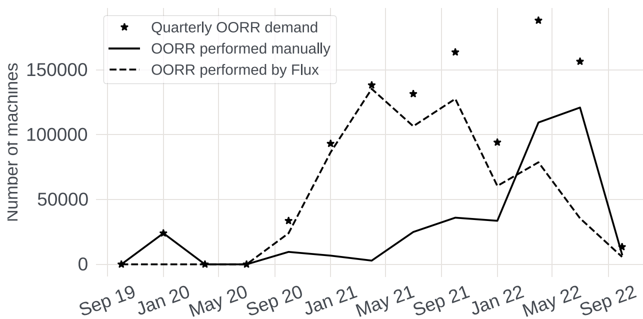

Figure 13: **Volume of out-of-region refresh (OORR).** This figure shows OORR planning and execution volumes handled by manual processes as well as Flux. The uptick in manual OORR in 2022 was due to a one-time large-scale decommission of data-warehouse hardware. Over all, Flux has helped scale our yearly OORR volume by ≈950%.

To understand Flux's capacity assignment, we focus on two sets of services: (1) the frontend `Web` service that serves as the traffic gateway for `FeatureStore`, and (2) all backend services that support or consume `FeatureStore`. Figure 15 shows the ratio of capacity for these two sets, which varies due to different workload mixes across regions. Specifically, regions A and B show a lower capacity ratio, because they are data-warehouse heavy and have a larger `FeatureStore` footprint to support additional training workloads that are co-located with data warehouse but do not go through `Web` to access `FeatureStore`. Before Flux was applied to `FeatureStore`, capacity planners needed to explicitly take this into consideration, whereas Flux's MIP formulation is able to automatically account for this and other factors affecting placement.

Recall from §3.3 that one optimization objective is to minimize deviation from the ratios in the globalized service model. This deviation is partially reflected in Figure 15 as the `Web-to-FeatureStore` capacity ratio's variances across regions. Before Flux was applied to `FeatureStore`, the variances across regions C, D, and E were partially due to suboptimal planning done by humans. Flux is able to reduce the variances by lowering the ratio for region D and increasing the ratio for region E, thus leading to better balance across regions C, D, and E. The improvement was small as this was the first time that Flux was applied to `FeatureStore` and was configured to be more conservative in introducing changes.

Deviation from ideal service capacity ratios is also reflected in the service's unbalanced CPU utilization across regions. Note that, if `FeatureStore`'s capacity increase in a region is bigger than `FeatureStore`'s traffic increase in the region, we expect `FeatureStore` to have a lower CPU utilization

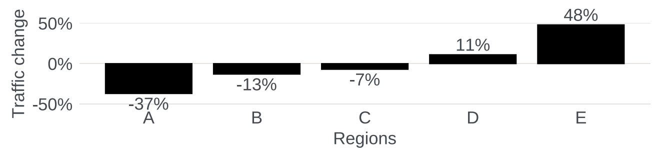

Figure 14: **`FeatureStore` traffic changes computed by Flux.** An example of how to read the figure: region A's traffic change is $\frac{\text{new traffic for FeatureStore in region A}}{\text{old traffic for FeatureStore in region A}}$-1=-37%. Only 5 out of 10s of regions are shown for readability.

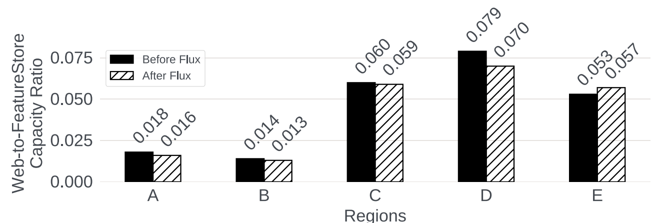

Figure 15: **Ratio of capacity for `Web` and `FeatureStore`.** Only 5 out of 10s of regions are shown for readability.

in the region after the change. This type of change can be applied to lower the CPU utilization of more heavily loaded regions, which is precisely what Flux did. For each region r, we calculate the average CPU utilization u_r of `FeatureStore` during its daily peak time window. Then we calculate the median of u_r across 10s of regions, and call it $\hat{u}$. Before and after Flux was applied to `FeatureStore`, $\hat{u}$ was 55% and 50%, respectively. This indicates that Flux is effective in matching traffic distribution with capacity distribution to balance load across regions.

Figure 16 shows u_r for some sample regions. Overall, Flux reduces the CPU utilization of more heavily loaded regions such as regions A and D. In this instance, Flux was unable to increase CPU utilization in C due to the other constraints and objectives. This example shows that there are many factors to be considered during optimization, which is better suited to a MIP solver than humans.

7 Related Work

Capacity management. Capacity management impacts service performance and reliability as well as an organization's capital and operating expenses. Two USENIX *;login:* articles [27, 48] provide an overview of this topic. Misbah et al. provide a survey of resource management in federated cloud [33]. Several publications report capacity-management practices for internet services such as LinkedIn [53, 56], Uber [15], Google [34], and Netflix [36]. They focus on forecasting demand and capacity headroom, and are complementary to our work focusing on global service placement.

Service tracing and modeling. Several systems [13,17,30,40, 46] insert unique request IDs into RPC calls to discover end-to-end service dependency. Our work adopts this approach. Some systems use statistical analysis [2] or heuristics [3, 38, 43] to infer service dependencies. Although these techniques are easier to deploy, they have not been proven to be robust enough to be used for internet-scale complex services. Both analytical models [51] and profiling techniques [41] have been applied to build performance models for three-tiered applications, but our environment is much more complex (see Figure 1). Endo *et al.* [20] call out the challenges of operating a distributed cloud, including resource modeling, but do not propose solutions.

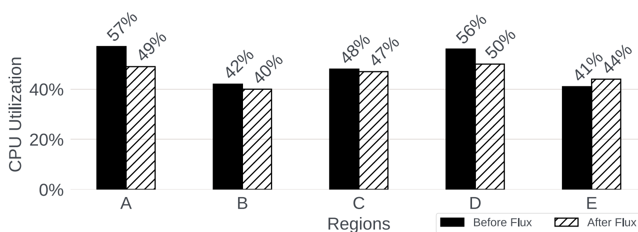

Figure 16: **CPU utilization of `FeatureStore`** per region before and after Flux is applied. Only 5 out of 10s of regions are shown for readability.

Service placement. Yang *et al.* [54] propose joint service placement and traffic routing in mobile cloud, without considering service interdependencies. Malet and Pietzuch [35] propose placing services across datacenters to minimize network latency without considering the constraints of capacity supply and demand.

Constrained optimization. Constrained optimization has been used for resource allocation in different scenarios, including hardware-to-reservation assignment within a region [37], data-shard-to-container placement [32], and job scheduling within a cluster [19,23–26,42,44,49,50]. None of them tackle the problem of global service placement.

Infrastructure orchestration. Cloud infrastructure orchestrators like Terraform [28] and CloudFormation [7] coordinate changes across multiple infrastructure systems in public clouds. These could be used to implement the orchestration component of Flux in a public cloud setting.

8 Conclusion

We identified the *regionalization problem* associated with managing customer services on large, global cloud footprints. We presented Flux, which solves regionalization by (1) using RPC tracing to build service regionalization models; (2) jointly solving service and traffic placement, growth capacity distribution, and infrastructure objectives; and (3) introducing a capacity orchestration system that safely and automatically rebalances services and traffic according to the computed plan. We shared our experience of using Flux at Meta's large private cloud and discussed ways in which the ideas in Flux can be applied to public clouds.

9 Acknowledgements

This paper presents many years worth of work by multiple teams at Meta. They include: the Regional Fluidity, Capacity Engineering, Capacity Automation, Shard Manager, and Algorithmic Optimization teams. We would like to thank the current members of the Flux team who are not already on the author list: Kiryong Ha, Alex Cauthen, Chris Zheng, Hossein Tajik, Lin Xiao, Xiaomeng Shen, Austin Hendy, Daniel Boeve, Jikai Zhang, Tejash Shah, Kevin Lin, Partha Roy Chowdhury, Caroline Tony, Junjie Qian, Anand Saggi, Sebastiano Peluso, David Xu, Yichen Zhou, and Peter John Daoud.

We would also like to thank the following for their insightful comments, honest feedback, and partnership during the development of Flux: Maria Kacik, James Kneeland, Nasser Manesh, Haying Wang, Ariel Rao, Ash Shroff, and Alp Elci.

Finally, we thank Yunqi Zhang, Dimitrios Skarlatos, all reviewers, and our shepherd Z. Morley Mao for their help and insightful feedback.

References

[1] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[2] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[3] Animashree Anandkumar, Chatschik Bisdikian, and Dakshi Agrawal. Tracking in a spaghetti bowl: monitoring transactions using footprints. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 133–144, 2008.

[4] Masoud Saeida Ardekani and Douglas B Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–381, 2014.

[5] Anonymized Authors. Meta's Geo-Replicated Pub-Sub Service. 2015. Research paper published at a conference.

[6] Anonymized Authors. Live traffic load testing in production at Meta. 2016. Research paper published at a conference.

[7] AWS. Provision Infrastructure As Code, 2021. https://aws.amazon.com/cloudformation/.

[8] AWS EC2. Use Capacity Rebalancing to handle Amazon EC2 Spot interruptions. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-capacity-rebalancing.html, 2022.

[9] AWS reserved instance, 2021. https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-reservation-models/amazon-ec2-reserved-instances.html.

[10] AWS user. Instance does not start—AWS out of capacity, 2016. https://answers.sap.com/questions/12184202/instance-does-not-start---aws-out-of-capacity.html.

[11] AWS user. Capacity shortage hits AWS UK micro instances, 2017. https://www.theregister.com/2017/03/24/aws_uk_t2_micro_instances_run_out/https://www.theregister.com/2017/03/24/aws_uk_t2_micro_instances_run_out/.

[12] AWS user. Hit with "insufficient capacity" for 3 days, 2018. https://www.reddit.com/r/aws/comments/97rnvj/hit_with_insufficient_capacity_for_3_days_do_i/.

[13] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[14] Daniel Boeve, Kiryong Ha, and Anca Agape. Throughput autoscaling: Dynamic sizing for Facebook.com, 2020. Blog post.

[15] Rick Boone. "Capacity Prediction" instead of "Capacity Planning" How Uber Uses ML to Accurately Forecast Resource Utilization, 2020. SREcon20 Americas, https://www.usenix.org/conference/srecon18americas/presentation/boone.

[16] Nathan Bronson, Zach Amsden, George Cabrera, 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. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[17] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[18] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, Kevin Cole, and Dmitri Perelman. Taiji: managing global user traffic for large-scale internet services at the edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 430–446, 2019.

[19] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, 2014.

[20] Patricia Takako Endo, Andre Vitor de Almeida Palhares, Nadilma Nunes Pereira, Glauco Estacio Goncalves, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mangs. Resource allocation for distributed cloud: concepts and research challenges. *IEEE network*, 25(4):42–46, 2011.

[21] Mary Jo Foley. Microsoft Azure customers reporting hitting virtual machine limits in U.S. East regions, 2019. https://www.zdnet.com/article/microsoft-azure-customers-reporting-hitting-virtual-machine-limits-in-u-s-east-regions/.

[22] Mary Jo Foley. European users reporting they're hitting Azure capacity constraints, 2020. https://www.zdnet.com/article/european-users-reporting-theyre-hitting-azure-capacity-constraints/.

[23] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[24] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[25] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[26] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, November 2016.

[27] David Hixson Kavita Guliani. Capacity Planning. *USENIX ;login:*, 40(1):49, 2015.

[28] HashiCorp. Terraform by HashiCorp, 2021. https://www.terraform.io/.

[29] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX, 2023.

[30] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50, 2017.

[31] Kripa Krishnan. Weathering the unexpected: Failures happen, and resilience drills help organizations prepare for them. *Queue*, 10(9):30–37, sep 2012.

[32] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[33] Misbah Liaqat, Victor Chang, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Toseef, Umar Shoaib, and Rana Liaqat Ali. Federated cloud resource management: Review and discussion. *Journal of Network and Computer Applications*, 77:87–105, 2017.

[34] Ramón Medrano Llamas. Capacity Planning at Scale, 2016. SREcon16 Europe, https://www.usenix.org/conference/srecon16europe/program/medrano-llamas.

[35] Barnaby Malet and Peter Pietzuch. Resource allocation across multiple cloud data centres. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, pages 1–6, 2010.

[36] Rajan Mittal and Andrew Park. Why Regional Reserved Instances Are a Game Changer for Netflix. In *AWS re:Invent*, 2017. https://www.youtube.com/watch?v=i1EW6zmFbSM.

[37] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[38] Patrick Reynolds, Janet L Wiener, Jeffrey C Mogul, Marcos K Aguilera, and Amin Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web*, pages 347–356, 2006.

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

[40] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver,

Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[41] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 71–84, 2005.

[42] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building scalable and flexible cluster managers using declarative programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[43] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N Chang. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *USENIX Annual technical conference*, 2009.

[44] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A Scalable Application Placement Controller for Enterprise Data Centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340, 2007.

[45] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.

[46] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):3–14, 2006.

[47] Tim Anderson. 'Azure appears to be full': UK punters complain of capacity issues on Microsoft's cloud, 2020. `https://www.theregister.com/2020/03/24/azure_seems_to_be_full/`.

[48] Luis Quesada Torres and Doug Colish. SRE Best Practices for Capacity Management. *USENIX ;login:*, 45(4):49, 2020.

[49] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[50] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[51] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):291–302, 2005.

[52] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[53] Ruoying Wang, Lei Zhang, Yang Yang, Yi Zhen, Bo Long, Tie Wang, Vinoth Govindaraj, Todd Palino, Samir Tata, and Viji Nair. CapPredictor: A Capacity Headroom Prediction Framework in Cloud. In *Workshop on Cloud Intelligence, associated with Artificial Intenlligence (AAAI 2020)*, 2020.

[54] Lei Yang, Jiannong Cao, Guanqing Liang, and Xu Han. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Transactions on Computers*, 65(5):1440–1452, 2015.

[55] Wei Zheng, Ricardo Bianchini, G John Janakiraman, Jose Renato Santos, and Yoshio Turner. Justrunit: Experiment-based management of virtualized data centers. In *Proc. USENIX Annual technical conference*, pages 18–18, 2009.

[56] Zhenyun Zhuang, Haricharan Ramachandra, Cuong Tran, Subbu Subramaniam, Chavdar Botev, Chaoyue Xiong, and Badri Sridharan. Capacity Planning and Headroom Analysis for Taming Database Replication Latency: Experiences with LinkedIn Internet Traffic. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 39–50, 2015.

S	Set of all services.
R	Set of all regions.
H	Set of all hardware types.
P	Set of all products.
T	Set of all timesteps.
t_0	Baseline timestep, i.e., infrastructure's current state.
t_1	Target timestep, i.e., the timestep for which we are executing.

Inputs

$m_{r,h,t}$	Capacity pool: amount of type h hardware available in region r at time t. It is generated by capacity forecast and includes the current capacity and net incoming and outgoing supply.
$\phi_{s,h,p,t}$	Globalized service baseline: output of the service-modeling process described in §3.2, indicating the fraction of service s' consumption of type h hardware being attributed to product p. We also scale the globalized service baseline by service growth, derived from our capacity budget management system, leading to varying values at different timesteps t.
$\gamma_{s,h,t}$	The total growth capacity, indicating the global amount of type h hardware allocated for service s at time t to support service growth. It is derived from Meta's budget management system, and is used to to support product growth and launches.
$\tau_{p,t}$	The total traffic growth indicating the global increase in traffic to product p at time t, represented as the percentage above above 100% global baseline traffic.
e_n	The penalty coefficient for objective n. Penalty coefficients dictate how tradeoffs are compared.

Assignment variables

Variable	Baseline	Distribution	Description
$x_{p,r,t}$	$\xi_{p,r}$	X_t	Traffic assignment, indicating the fraction of traffic from Regionalization Entity e assigned to region r at time t. $X_t := \{x_{p,r,t} : p \in P, r \in R\}$ Invariant: $\forall p \in P \quad \sum_{r\in R} \xi_{p,r} = 1$
$c_{s,r,h,t}$	$\kappa_{s,r,h}$	C_t	Capacity assignment, indicating the amount of type h hardware allocated to service s in region r at time t. $C_t := \{c_{s,r,h,t} : s \in S, r \in R, h \in H\}$
$g_{s,r,h,t}$	-	G_t	Growth assignment, indicating the amount of additional type h hardware allocated to service s in region r at time t for the purpose of growth. $G_t := \{g_{s,r,h,t} : s \in S, r \in R, h \in H\}$
$d_{r,h,t}$	-	D_t	Deficit assignment, indicating the amount of additional type h hardware needed in region r at time t. $D_t := \{d_{r,h,t} : r \in R, h \in H\}$
$s_{r,h,t}$	-	S_t	Spares, indicating the amount of unallocated hardware of type h in region r at timestep t. $S_t := \{s_{r,h,t} : r \in R, h \in H\}$
$o_{r,h,t}$	-	O_t	Double occupancy capacity. See explanation for Expression 14. $O_t := \{o_{r,h,t} : r \in R, h \in H\}$
$r_{s,r,h,t}$	$\rho_{s,r,h}$	-	Model residual, the difference between the observed baseline capacity distribution and the capacity distribution implied by the globalized service baseline, ϕ, distributed according to the baseline traffic distribution ξ. $\rho_{s,r,h} := \kappa_{s,r,h} - \sum_{p\in P} \xi_{p,r} \times \phi_{s,h,p,t_0}$

Table 1: **Notation used in the MIP formulation.** *Baseline* means the current state of the infrastructure.

A MIP Formulation in Flux

This appendix presents the MIP formulation used by Flux.

The core of the formulation is an assignment problem, represented by the assignment variables enumerated in Table 1. Each variable shares a region (r) and timestep (t) dimension; while capacity related assignments also include dimensions for the service being assigned (s) and hardware type of the assignment (t).

Next, we present the MIP problem formulation and explain each expression. The MIP problem is to minimize:

$$e_1 \times \sum_{p\in P, r\in R, t\in T} |x_{p,r,t} - x_{e,r,t_0}| \tag{2}$$

$$+ e_2 \times \sum_{t\in T, p\in P} \max_{r\in R} x_{p,r,t} \tag{3}$$

$$+ e_3 \times \sum_{r\in R, h\in H, t\in T} d_{r,h,t} \tag{4}$$

$$+ e_4 \times \sum_{r\in R, h\in H, t\in T} o_{r,h,t} \tag{5}$$

$$+ e_5 \times \sum_{p\in P, r\in R, h\in H, t\in T} |r_{p,r,h,t}| \tag{6}$$

$$+ e_6 \times \sum_{r\in R} \left| s_{r,h,t} - \frac{s_{r,h,t}}{\sum_{r\in R} s_{r,h,t}} \right| \quad \forall h \in H, t \in T \tag{7}$$

Subject to:

$$c_{s,r,h,t_0} = \kappa_{s,r,h} \quad r_{s,r,h,t_0} = \rho_{s,r,h,t_0}$$

$$x_{p,r,t_0} = \xi_{e,r} \quad o_{r,h,t_0} = 0 \tag{8}$$

$$c_{s,r,h,t} \geq \sum_{p\in P} (x_{p,r,t} \times \phi_{s,h,p,t}) + r_{s,r,h,t} \tag{9}$$

$$r_{s,r,h,t} \geq \min\{\rho_{s,r,h}, 0\} \tag{10}$$

$$m_{r,h,t} + d_{r,h,t} = s_{r,h,t} + \sum_{s\in S} c_{s,r,h,t} + \sum_{s\in S} g_{s,r,h,t} \tag{11}$$

$$\forall p \in P, t \in T \quad \sum_{r\in R} x_{p,r,t} = 1 + \tau_{p,t} \tag{12}$$

$$g_{s,r,h,t} = \frac{\gamma_{s,h,t} * c_{s,r,h,t}}{\sum_{r\in R} c_{s,r,h,t}} \tag{13}$$

$$o_{r,h,t} = \sum_{s\in S} c_{s,r,h,t-1} - c_{s,r,h,t} [c_{s,r,h,t} < c_{s,r,h,t-1}] \tag{14}$$

$$s_{r,h,t} + \sum_{s\in S} g_{s,r,h,t} \geq o_{r,h,t} \tag{15}$$

Below, we explain the intuition behind the MIP expressions.

Stability objective. Expression 2 penalizes traffic shifts to reduce churn in the infrastructure.

Disaster-readiness objective. Services have a disaster-readiness buffer to cope with any single-region failure. Expression 3 minimizes this buffer, by minimizing the size of the largest region.

Objective to minimize deficits. Expression 4 minimizes the additional hardware needed. Technically, a feasible solution requires that $\forall r \in R, h \in H, t \in T \quad d_{r,h,t} = 0$, and we use a high penalty p_4 to ensure that deficits are non-zero only if a solution requires it.

Objective to minimize deviation from service model. Pre-existing placement imbalance, attribution inaccuracies, or traffic routing imbalances can cause `baseline` capacity assignments to deviate from the service model. Expression 6 minimizes deviation from the global service model defined by the globalized service baseline. $r_{p,r,h,t}$ is the residual of the service model, which is further discussed in §6.3.

Objective to balance spare pool distribution Expression 7 encourages balancing unused capacity across regions to reduce hardware stranding. Sufficient unused capacity placement can also act as a buffer for capacity estimation discrepancies.

Baseline. Expression 8 establishes timestep t_0 as the `baseline`, i.e., the current state of the infrastructure.

Product attribution ratio constraint. Expression 9 ensures that each service is allocated according to the ratio imputed from the service model (see §3.2). The model residual $r_{s,r,h,t}$ is used to offset the placement.

Residual-regression constraint. Expression 10 prevents model residuals from regressing. Specifically, negative residuals (i.e., underprovisioned services per the model) are prevented from becoming more negative, while positive residuals may not become negative. Together, objective 6 and this constraint cause Flux to better balance service utilization across regions. Flux provides fine-grained controls (per service, region, hardware type) that let the service owner tune how aggressively Flux is allowed to rebalance a service. §6.5 provides a case study of this benefit.

Capacity-sufficiency constraint. Expression 11 ensures that each region has sufficient capacity to support the capacity assignment, as determined by expression 9. This also assigns additional hardware as deficits, if required for feasibility. Unallocated capacity is assigned to the spare pool S_t.

Full-placement constraint. Expression 12 ensures that each RE is fully placed.

Organic growth constraint. When $\tau_{p,t} \geq 0$ Expression 12 places additional traffic demand which Expression 9 then allocates the organic growth capacity to each service according to its globalized service model. Organic growth is used to model increased traffic where all dependent services much be sized up proportionally.

Inorganic growth constraint. Expression 13 distributes the growth capacity proportionally to a service's placed capacity. Inorganic growth is used to distribute growth capacity to an individual service such that dependent services don't necessarily need to be resized, such as product launches.

Double-occupancy constraint. We execute the capacity upsizes before capacity downsizes (see §3.4) and cannot count on future released capacity to fund an ongoing upsize operation. Expression 14 defines the amount of capacity needed during an upsize operation, whereas Expression 15 ensures a valid intermediate state by enforcing sufficient capacity is available to prevent an upsize from using still occupied capacity. Growth is given out as a final step of execution, and can be used during the upsize stage. Expression 5 minimizes double occupancy.

MIP Solver Scalability. Flux uses the MIP solver well within its scalability limit. Our latest service placement run generated a problem with 24K assignment variables and 36K constraints. Solving the problem took only 5 seconds.