Twine: A Unified Cluster Management System for Shared Infrastructure

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 Peter Zhang

Facebook Inc.

Abstract

We present *Twine*, Facebook's cluster management system which has been running in production for the past decade. Twine has helped convert our infrastructure from a collection of siloed pools of customized machines dedicated to individual workloads, into a large-scale shared infrastructure with fungible hardware.

Our goal of ubiquitous shared infrastructure leads us to some decisions counter to common practices. For instance, rather than deploying an isolated control plane per cluster, Twine scales a single control plane to manage one million machines across all data centers in a geographic region and transparently move jobs across clusters.

Twine accommodates workload-specific customization in shared infrastructure, and this approach further departs from common practices. The *TaskControl* API allows an application to collaborate with Twine to handle container lifecycle events, e.g., restarting a ZooKeeper deployment's followers first and its leader last during a rolling upgrade. *Host profiles* capture hardware and OS settings that workloads can tune to improve performance and reliability; Twine dynamically allocates machines to workloads and switches host profiles accordingly.

Finally, going against the conventional wisdom of prioritizing stacking workloads on big machines to increase utilization, we universally deploy power-efficient small machines outfit with a single CPU and 64GB RAM to achieve higher performance per watt, and we leverage autoscaling to improve machine utilization.

We describe the design of Twine and share our experience in migrating Facebook's workloads onto shared infrastructure.

1 Introduction

The advent of computation as a utility has led organizations to consolidate their workloads onto *shared infrastructure*, a common pool of resources to run any workload. Cluster management systems help organizations utilize shared infrastructure effectively through automation, standardization, and economies of scale. Cluster management systems have made large progress in the past decade, from Mesos [17], Borg [39], to Kubernetes [23]. Existing systems, however, still have limitations in supporting large-scale shared infrastructure:

1. They usually focus on isolated clusters, with limited support for cross-cluster management as an afterthought. These silos may strand unused capacity in clusters.
2. They rarely consult an application about its lifecycle management operations, making it more difficult for the application to uphold its availability. For example, they may unknowingly restart an application before it has built another data replica, rendering the data unavailable.
3. They rarely allow an application to provide its preferred custom hardware and OS settings to shared machines. Lack of customization may negatively impact application performance on shared infrastructure.
4. They usually prefer big machines with more CPUs and memory in order to stack workloads and increase utilization. If not managed well, underutilized big machines waste power, often a constrained resource in data centers.

These limitations can lead to underdelivery of the promise of shared infrastructure: (1) artificially caps the sharing scope to one cluster; (2) & (3) highlight the tension between shared infrastructure's preference for standardization and applications' needs for customization; (4) calls for a shift of focus from single-machine utilization to global optimization.

In this paper, we describe how we address the above limitations in *Twine*, Facebook's cluster management system. Our two insights are 1) we scale a single Twine control plane to manage one million machines across data centers in a geographic region while providing high reliability and performance guarantees, and 2) we support workload-specific customization, which allows applications to run on shared infrastructure without sacrificing performance or capabilities.

Twine packages applications into Linux containers and manages the lifecycle of machines, containers, and applications. A *task* is one instance of an application deployed in a container, and a *job* is a group of tasks of the same application.

A single control plane to manage one million machines. A region consists of multiple data centers, and a data center is usually divided into *clusters* of tens of thousands of machines connected by a high-bandwidth network. As with Borg [39] and Kubernetes [23], an isolated control plane per cluster results in stranded capacity and operational burden because workloads cannot easily move across clusters. For example, power-hungry jobs colocated in a cluster can trigger power capping [26, 41], affecting service throughput until humans move the problematic jobs to other clusters.

Similarly, large-scale hardware refresh in a cluster may result in idle machines and operational overhead. Our current hardware refresh granularity is 25% of a data center. Figure 1 shows the duration for all owners of thousands of jobs to migrate jobs out of a cluster prior to a hardware refresh in 2016. The P50 is at 7.5 days and the P100 is at 87 days. A large portion of the cluster sat idle in these ≈80 days while waiting for all jobs to migrate.

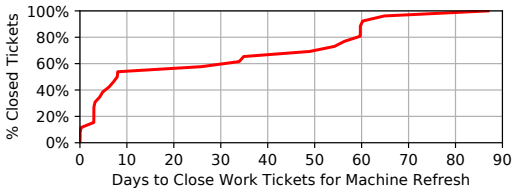


Figure 1: CDF of time to close job-migration work tickets.

To address the problems above, we scaled a single Twine control plane to manage one million machines across all data centers in a region. Unlike Kubernetes Federation [25], Twine scales out natively without an additional federation layer.

Collaborative lifecycle management. Cluster management systems generally lack visibility into how an application manages its internal state, leading to suboptimal handling of hardware and software lifecycle events that impact application availability. Figure 2 provides a stateful service example.

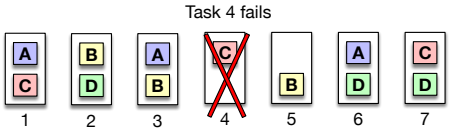


Figure 2: Replicas of data shards `A`–`D` are distributed across tasks 1-7. Tasks 1 and 3 should not be restarted concurrently for a software upgrade, as shard `A` would lose two replicas and become unavailable. If the machine hosting task 4 were to fail or be restarted for a kernel or firmware upgrade, the cluster management system would need to ensure that neither task 1 nor 7 is restarted concurrently in order to keep shard `C` available.

Twine provides a novel *TaskControl* API to allow applications to collaborate with Twine in handling task lifecycle events that impact availability. For example, an application may postpone a task restart and rebuild a lost data replica first.

Host-level customization. Hardware and OS settings may significantly impact application performance. For example, our web tier achieves 11% higher throughput by tuning OS settings. Twine leverages *entitlements*, our quota system, to handle hardware and OS tuning. For example, an entitlement for a business unit may allow it to use up to 30,000 machines. We associate each entitlement with a *host profile*, a set of host customizations that the entitlement owner can tune. Out of a shared machine pool, Twine dynamically allocates machines to entitlements and switches host profiles accordingly.

Power-efficient machines. Facebook's workloads have grown faster than our data center buildup. Power scarcity motivated us to maximize performance per watt, either by employing universal stacking on big machines or deploying power-efficient small machines. We found it challenging to stack large workloads on big machines effectively. Further, unlike a public cloud that needs to support diverse customer requirements, we only need to optimize for our internal workloads. These factors led to us to adopt small machines with a single CPU and 64GB RAM [32].

Shared infrastructure. As we evolved Twine to support large-scale shared infrastructure, we have been migrating our workloads onto a single shared compute pool, *twshared*, and a single shared storage pool. Twine supports both pools, but we focus on twshared in this paper. twshared hosts thousands of systems, including frontend, backend, ML, stream processing, and stateful services. While twshared does not host durable storage systems, it provides TBs of local flash to support stateful services that store state derived from durable storage systems. Figure 3 shows twshared's growth.

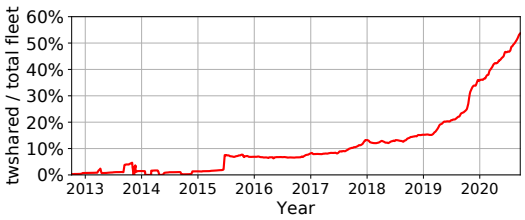


Figure 3: Growth of twshared. twshared was created in 2013, but adoption was limited in its first six years. We enhanced Twine and rebooted the adoption effort in 2018. twshared hosts 56% of our fleet as of October 2020, in contrast to 15% in January 2019. We expect that all compute services, ≈85% of our fleet, will run on twshared by early 2022, while the remaining 15% will run in a separate shared storage pool.

twshared has become our ubiquitous compute pool, as all new compute capacity lands only in twshared. We had broad conversations with colleagues in industry and are unaware of any large company that has achieved near 100% shared infrastructure consolidation.

The rest of the paper is organized as follow. §2 presents the design and implementation of Twine. §3 and §4 describe how we scale Twine to manage one million machines and do so reliably. §5 evaluates Twine. §6 shares our experience with driving twshared adoption. §7 describes lessons learned. §8 summarizes related work. Finally, §9 concludes the paper.

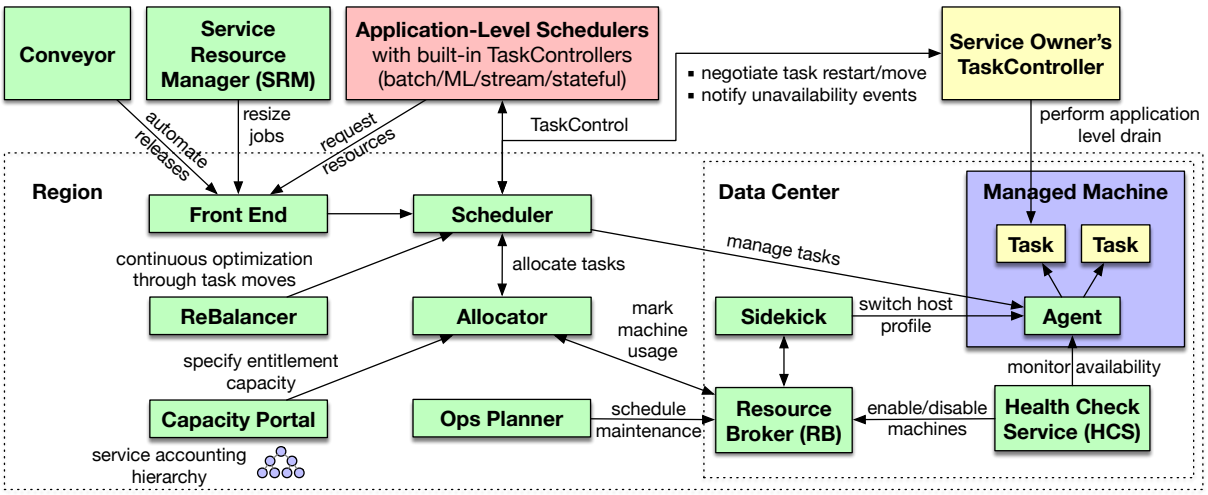


Figure 4: The Twine Ecosystem. Note a potential terminology confusion. The Twine *scheduler* corresponds to the Kubernetes [23] controllers, whereas the Twine allocator corresponds to the Kubernetes scheduler.

2 Twine Design and Implementation

Facebook currently operates out of 12 geo-distributed *regions*, with several more under construction. Each region consists of multiple *data center* (DC) buildings. A *main switchboard* (MSB) [41] is the largest fault domain in a DC with sufficient power and network isolation to fail independently. A DC consists of tens of MSBs each powering tens of rows that feed tens of racks of servers as shown in Figure 5.

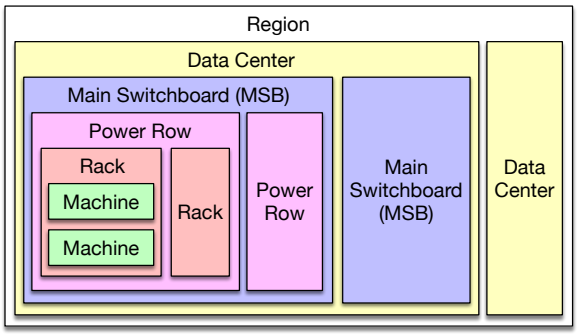


Figure 5: Data center topology.

Historically, a *cluster* was a subunit within a DC consisting of about ten thousand machines connected by a high-bandwidth network and managed by an isolated Twine control plane. Over time, our network transitioned to a fabric architecture [2, 14] that provides high bandwidth both within a DC and across DCs in a region, empowering a single Twine control plane to manage jobs across DCs.

2.1 Twine Ecosystem

Figure 4 shows an overview of Twine. The *Capacity Portal* allows users to request or modify *entitlements*, which associate capacity quotas with business units defined in the *service accounting hierarchy*. With a granted entitlement, a user deploys jobs through the *front end*. The *scheduler* manages job and task lifecycle, e.g., orchestrating a job's software release. If a job has a *TaskController*, the scheduler coordinates with the TaskController to make decisions, e.g., delaying a task restart to rebuild a lost data replica first. The *allocator* assigns machines to entitlements and assigns tasks to machines. *ReBalancer* runs asynchronously and continuously to improve the allocator's decisions, e.g., better balancing the utilization of CPU, power, and network. *Resource Broker* (RB) stores machine information and *unavailability events* that track hardware failures and planned maintenance. DC operators schedule planned maintenance through *Ops Planner*. The *Health Check Service* (HCS) monitors machines and updates their status in RB. The *agent* runs on every machine to manage tasks. *Sidekick* switches host profiles as needed. *Service Resource Manager* (SRM) autoscales jobs in response to load changes. *Conveyor* is our continuous delivery system.

2.2 Entitlements

Conceptually, an *entitlement* is a pseudo cluster that uses a set of dynamically allocated machines to host jobs. An entitlement grants a business unit a quota expressed as a count of machines of certain types (e.g., 2,000 Skylake machines) or as Relative Resource Units (RRU) akin to ECU in AWS.

A machine is either free or assigned to an entitlement, and it can be dynamically reassigned from one entitlement to another. An entitlement can consist of machines from different DCs in a region. Existing cluster management systems bind a job to a physical cluster. In contrast, Twine binds a job to an entitlement. Jobs in an entitlement stack with one another on machines assigned to the entitlement.

By default, Twine spreads tasks of the same job across DCs and MSBs as shown in Figure 6. This reduces buffer capacity

needed for fault tolerance [29]. Suppose a job's tasks are spread across 12 MSBs in one DC. We need $\frac{1}{12} \approx 8.3\%$ of buffer capacity to guard against the failure of one MSB. If the job's tasks are spread across five DCs' 60 MSBs, the needed buffer reduces to $\frac{1}{60} \approx 1.7\%$. For workloads that require better locality for compute and storage, Twine allows an entitlement to override the default spread policy and pin its machines and jobs to a specific DC. These workloads are in the minority.

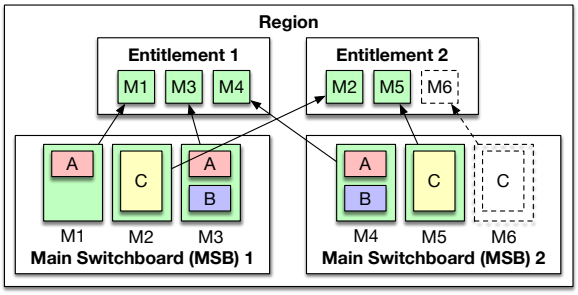


Figure 6: Entitlement example. `Entitlement 1` consists of machines `M1`, `M3`, and `M4` from different MSBs. Jobs `A` and `B` are bound to `Entitlement 1`, and job `C` is bound to `Entitlement 2`. Jobs `A` and `B` stack their tasks on machine `M3`. As job `C` grows, Twine adds machine `M6` to `Entitlement 2`.

The allocator assigns machines to entitlements, and it also assigns tasks to machines in an entitlement. For an entitlement with a quota of N machines, the number of machines actually assigned to the entitlement may vary between 0 and N, depending on the actual needs of jobs running in the entitlement. Figure 7 depicts an example of how an entitlement changes over time.

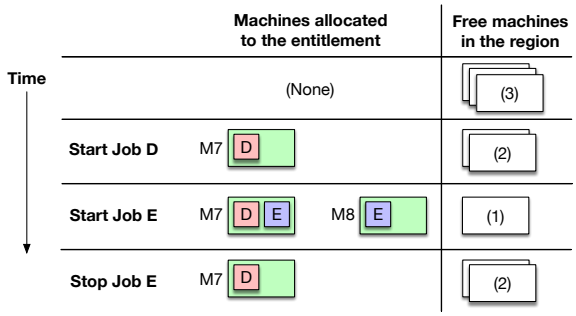


Figure 7: Allocation of machines and tasks. Initially, no machine is assigned to the entitlement. When job `D` starts, the allocator assigns machine `M7` to the entitlement. When job `E` starts, the allocator stacks one task on `M7` and adds machine `M8` to the entitlement to run `E`'s other task. When job `E` stops, the allocator returns `M8` to the free machine pool for use by other entitlements.

We optimized the allocator to make quick decisions when starting tasks; this optimization limits computation time and leads to best-effort outcomes. The addition or removal of machines and workload evolution may result in hotspots in CPU, power, or network. ReBalancer runs asynchronously and continuously to improve upon the allocator's allocation decisions by swapping machines across entitlements or moving tasks across machines. ReBalancer uses a constraint solver to perform these time-consuming global optimizations.

Entitlements help automate job movements across clusters. Consider a cluster-wide hardware refresh. We first add new machines from other clusters into the regional free machine pool (see the right side of Figure 7). Then the allocator moves tasks from machines undergoing hardware refresh to new machines acquired from the free machine pool, requiring no actions from the job owner. To migrate a task, Twine stops the task on the old machine and restarts it on the new machine. We do not use live container migration.

2.3 Allocator

One instance of Resource Broker (RB) is deployed to each DC. RB records whether a machine in the DC is free or assigned to an entitlement. A regional allocator fetches this information from all RBs in the same region, maintains an in-memory write-through cache, and subscribes to future changes.

The scheduler calls the allocator to perform a job allocation when a new job starts, an existing job changes size, or a machine fails. The allocation request contains an entitlement ID, an allocation policy, and a per-task map of which tasks need to be allocated or freed. The allocation policy includes hard requirements (e.g., using Skylake machines only) and soft preferences (e.g., spreading tasks across fault domains).

The allocator maintains an in-memory index of all machines and their properties to support hard requirement queries, such as "*all Skylake machines with available CPU $\geq$ 2RRU and available memory $\geq$ 5GB*." It needs to search machines beyond the ones already assigned to the entitlement because it may need to add more machines to the entitlement to host the job. After applying hard requirements, it applies soft preferences to sort the remaining machines.

A soft preference is expressed as a combination of 1) a machine property to partition machines into different bins with the same property value, and 2) a strategy to allocate tasks to these machine bins. For example, the allocator spreads tasks across fault domains by using a soft preference with `fault domain` as the machine property, and the strategy that assigns tasks evenly to the machine bins that represent fault domains.

The allocator uses multiple threads to perform concurrent allocations for different jobs, and relies on optimistic concurrency control to resolve conflicts. Before committing an allocation, a thread verifies that all impacted machines still have sufficient resources left for the allocation. If the verification fails, it retries a different allocation.

To avoid repeating the costly machine selection process, the allocator caches the allocation results at the job level. The allocator invalidates a cache entry if the job allocation request changes or the properties of the machines hosting the tasks change. The cache hit ratio is typically above 99%.

2.4 Scheduler

The scheduler manages the lifecycle of jobs and tasks. As the central orchestrator, the scheduler drives changes across

Twine components in response to different lifecycle events, including hardware failures, maintenance operations, power capping [41], kernel upgrades, job software releases, job resizing, task canary, and ReBalancer moving tasks.

The scheduler handles a machine failure as follows. When the Health Check Service detects a machine failure, it creates an unavailability event in Resource Broker, which notifies the allocator and scheduler. The scheduler disables the affected tasks in the service discovery system so that clients stop sending traffic to these tasks. A job is impacted by the machine failure if it has tasks running on the machine. If an impacted job has a TaskController, the scheduler informs the TaskController of the affected tasks. After the TaskController acknowledges that these tasks can be moved, the scheduler requests the allocator to deallocate the tasks and allocate new instances of the tasks on other machines. The scheduler instructs agents to start the new tasks accordingly. Finally, the scheduler enables the tasks in the service discovery system so that clients can send traffic to the newly started tasks.

The scheduler paces changes to a job's tasks to avoid application downtime. For example, regardless of reasons (e.g., hardware failure or software upgrade), if a job's total unavailable tasks exceed a user-configured threshold, no more tasks can be restarted for a software release. The scheduler has built-in support for commonly used lifecycle policies and offers the TaskControl API to implement more complex policies.

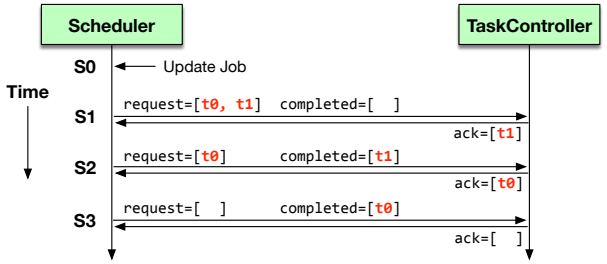
2.5 TaskControl

An application often knows best how to safely handle hardware or software lifecycle events that affect its availability, but it cannot inform the cluster management system how to orchestrate these actions. Figure 2 depicts one example. Another example is a ZooKeeper deployment that wishes to apply a software release to its followers first and its leader last [8]. Otherwise, an n-member ZooKeeper ensemble in the worst case experiences n leader failovers during a release. We designed the *TaskControl* API to allow applications to collaborate with Twine when deciding which task operations to proceed and which to postpone, as depicted in Figure 8.

Unlike software releases, maintenance events like a power device replacement cannot be blocked indefinitely by a TaskController; the scheduler gives the TaskController advance notices with a deadline to react. Upon reaching the deadline, the scheduler stops the remaining tasks on the affected machines, allowing maintenance to proceed. Before the deadline, a TaskController has multiple options: 1) move the tasks to other machines, 2) stop the tasks on the current machine and restart them after the maintenance completes, or 3) do nothing and keep the tasks running. For example, a top-of-rack switch maintenance typically incurs only a few minutes of network downtime, and a stateful service may prefer option 3 because rebuilding a data replica elsewhere takes longer than the maintenance itself.

```
service TaskController {
  TaskControlResponse process(TaskControlRequest request);
}
struct TaskControlRequest {
  string jobHandle;
  list<> request; // Pending task operations to be approved.
  list<> completed; // Completed task operations.
  list<> advanceNotices; // Upcoming planned maintenance.
  list<> allUnhealthyTasks; // Tasks unhealthy due to any reason.
  int sequenceNumber; // Increase after each call.
}
struct TaskControlResponse {
  list<> ack; // Approved task operations.
}
```

(a) TaskControl API.



(b) Calling sequence of the TaskControl API when handling a job update. The job has two tasks: `t0` and `t1`. At time `S0`, the user initiates a job update. At time `S1`, the scheduler requests the approval of updates on tasks `t0` and `t1`, with `request=[t0,t1]`. The application's TaskController can selectively approve updates for any subset of tasks in any order. It approves the update on task `t1` by replying `ack=[t1]`, but delays the update on task `t0` to keep one task available. At time `S2`, the scheduler completes the update on task `t1` with `completed=[t1]`, and requests an update on the remaining task `t0`. This time, the TaskController approves the request.

Figure 8: TaskControl API and an example of the calling sequence.

2.6 Host Profiles

Our fleet runs thousands of different services, and Figure 9 shows that the 50 largest services consume ≈70% of all capacity. Similar capacity skew exists in Borg as well [36].

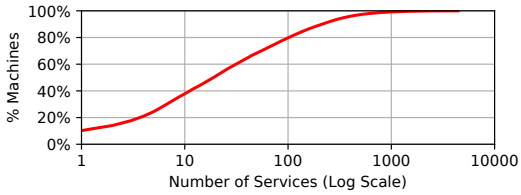


Figure 9: CDF of machines used by services. A small number of services dominate the capacity consumption. Note that the x-axis is in log scale.

Our efficiency effort focuses on these large services, and we find that host-level customization is important for maximizing their performance. For example, customizations help our large web tier achieve 11% higher throughput. However, some custom settings may be beneficial for one service but detrimental to another. As an example, a combination of explicit 2MB and 1GB hugepages improves the web tier's throughput by 4%; however, most services are incapable of utilizing explicit hugepages and enabling this setting globally would lead to unusable memory.

We resolved the conflict between host-level customization and sharing machines in a common pool via *host profiles*, a framework to control host-level customizations on entitlements. An entitlement is associated with one host profile; all machines in the entitlement share the same host profile. When a machine is reassigned from one entitlement to another, *Sidekick* automatically applies the target entitlement's host profile. By fully automating the process of machine allocation and host customization in our shared infrastructure, we can perform fleet-wide optimizations (e.g., swapping machines across entitlements to eliminate hotspots in network or power) without sacrificing workload performance. Supported host profile settings include kernel versions, sysctls (e.g., hugepages and kernel scheduler settings), cgroupv2 (e.g., CPU controller), storage (e.g., XFS or brtfs), NIC settings, CPU Turbo Boost, and hardware prefetch.

2.7 Application-Level Schedulers

As shown at the top of Figure 4, multiple application-level schedulers are built atop Twine to better support vertical workloads such as stateful [16], batch [21], machine learning [13], stream processing [28], and video processing [18]. Twine provides containers as resources for these application-level schedulers to manage and delegates task lifecycle management to them through TaskControl.

Shard Manager (SM) [16] is an example of an application-level scheduler. It is widely used at Facebook to build sharded services like the one in Figure 2. It has two major components: the SM client library and the SM scheduler. The library is linked into a sharded service and provides two APIs for the service to implement: `add_shard()` and `drop_shard()`. The SM scheduler decides the shards each Twine task will host and calls the service's `add_shard()` implementation to prepare the task to serve requests for those shards. To balance load, SM may migrate a shard from task T_1 to task T_2 by informing T_1 to `drop_shard()` and T_2 to `add_shard()`.

The SM scheduler integrates with Twine through TaskControl and can handle the complex situations depicted in Figure 2. In another example, Twine gives SM advance notice about an upcoming maintenance on a machine. If the maintenance duration is short and the shards hosted by the machine have replicas elsewhere, SM may do nothing; otherwise, SM may migrate the impacted shards out of the machine.

2.8 Small Machines and Autoscaling

To achieve higher performance per watt, our server fleet uses millions of small machines [32], each with one 18-core CPU and 64GB RAM. We have worked with Intel to define low-power processors optimized for our environment, e.g., removing unneeded NUMA components. Four small machines are tightly packed into one sled, sharing one multi-host NIC. They are replacing our big machines, each with dual CPUs, 256GB RAM, and a dedicated NIC. Under the same rack-level power budget, a rack holds either 92 small machines or 30 big machines. A small-machine rack delivers 57% higher total compute capacity measured in RRU. Averaged across all our services, using small machines led to 18% savings in power and 17% savings in total cost of ownership (§5.4).

We are consolidating all our compute services onto small machines, as opposed to offering a variety of high-memory or high-CPU machine types. This unification simplifies downstream supply chain and fleet management. It also improves machine fungibility across services, as we can easily reuse a machine across all compute services. Our consolidation journey has been challenging (§7.4), as some services initially did not fit the limited 64GB in our small machines. To address this, we used several common software architectural changes:

- Shard a service so that each instance consumes less memory. Our Shard Manager platform (§2.7) helps developers easily build sharded services running on Twine.
- Exploit data locality to move in-memory data to an external database and use the smaller memory as a cache.
- Exploit data locality to provide tiered memory on top of 64GB RAM and TBs of local flash. For example, when migrating TAO [7], our social graph cache, from big machines to small machines, CacheLib [5] transparently provided tiered memory to improve cache hit ratio and reduce load on the external database by ≈30%.

Our largest services fully utilize small machines without stacking. We rely on Autoscaling to free up underutilized machines. Active Last Minute (ALM) is the number of people who use our online products within a one-minute interval. The load of many services correlates with ALM. *Service Resource Manager* (SRM) uses historical data and realtime measurements to continuously adjust task count for ALM-tracking services and frees up underutilized machines in their entirety for other workloads to use. This work has allowed us to successfully build a large-scale shared infrastructure that consists primarily of small machines.

3 Scaling to One Million Machines

We designed Twine to manage all machines that can fit in a region's 150MW power budget. Although none of our regions host one million machines yet, we are close and anticipate reaching that scale in the near future. Two principles help Twine scale to one million machines and beyond: 1) sharding as opposed to federation, and 2) separation of concerns.

3.1 Scale Out via Sharding

To scale out, we shard Twine schedulers by entitlements, as depicted in Figure 10. We assign newly-created entitlements to shards with the least load. Entitlements can change size and can migrate across shards. If a shard becomes overloaded,

Twine can transparently move an entitlement in the shard to another shard without restarting tasks in the entitlement. Twine can also migrate an individual job from one entitlement to another. To do this, Twine performs a rolling update of the job until all tasks restart on machines belonging to the new entitlement. We automate the execution of these migrations, but humans still decide when and what to migrate. Since migrations happen rarely, we do not automate these further.

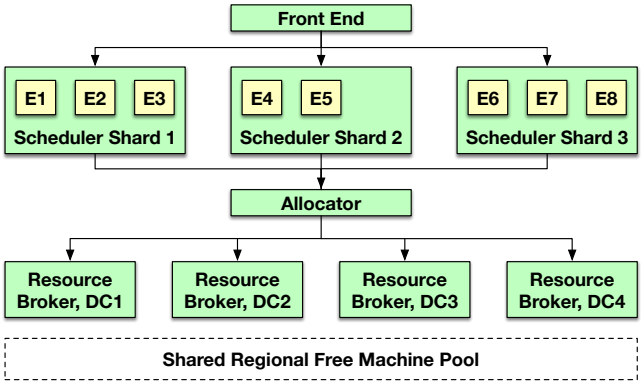


Figure 10: Sharding a scheduler by entitlements. Each scheduler shard manages a different subset of entitlements for the region. `Scheduler Shard 1` manages entitlements `E1`, `E2`, and `E3`. The front end maintains an entitlement-to-shard map and forwards requests to the responsible shards. Each data center has a Resource Broker (RB) managing the machines in that data center. Conceptually, all RBs in a region jointly maintain a free machine pool shared by all entitlements in the region. We also shard the allocator by entitlements and there is a 1:1 mapping between a scheduler shard and an allocator shard. We do not show allocator sharding in the figure as it currently manages a small fraction of our fleet and is still in the process of broader production deployment.

With sharding, the scheduler can easily scale to one million machines. Each data point in Figure 11 plots the P99 CPU utilization of a scheduler shard. The largest shard manages ≈170K machines, using up to 40 cores and 80GB memory. We are moving towards smaller shards to reduce the impact of a shard failure. Assuming each shard manages 50K machines in the future, a single Twine deployment can manage 1M machines with 20 shards. We believe Twine can easily scale beyond 1M machines by adding more shards.

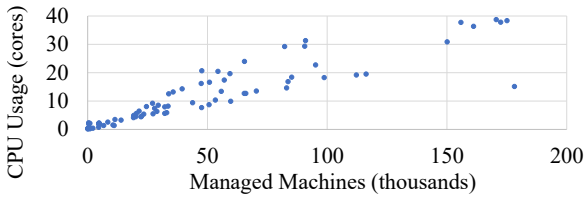


Figure 11: P99 CPU usage of production scheduler shards over one week.

The simplicity of scheduler sharding comes with a theoretical limitation: a single job must fit in a single scheduler shard. This is not a practical limitation. Currently, the largest scheduler shard manages ≈170K machines; the largest entitlement uses ≈60K machines; and the largest job has ≈15K tasks.

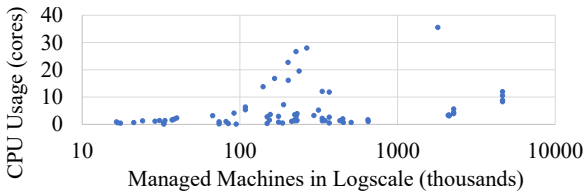


Figure 12: P99 CPU usage of production allocators over one week.

Each data point in Figure 12 plots the P99 CPU utilization of a production allocator. At its peak, a large allocator performs ≈1,000 job allocations per second, with an average job size of 36 tasks. We run a few deployments of the scheduler and allocator at the global level to manage machines and jobs across multiple regions (§7.3). Our largest global allocator currently manages more than one million machines across regions. The allocator is scalable because it has a high cache hit ratio (§2.3), does not handle allocations for short-lived batch jobs (§3.2), and does not perform time-consuming optimizations (§3.2).

3.2 Scale Out via Separation of Concerns

We avoid Kubernetes' centralized architecture where all components interact through one central API server and share one persistent store. These centralized components become bottlenecks and limit Kubernetes' scalability to 5K machines. We shard all Twine components and scale them out independently. Sharded components include the front end, scheduler, allocator, Resource Broker, Health Check Service, and Sidekick. Further, each stateful Twine component (front end, scheduler, allocator, and RB) has its own separate persistent store for metadata. Like Kubernetes [23] and unlike Borg [39], we use external persistent stores for components, as opposed to building the stores directly into components. This allows us to independently shard and scale out persistent stores as needed.

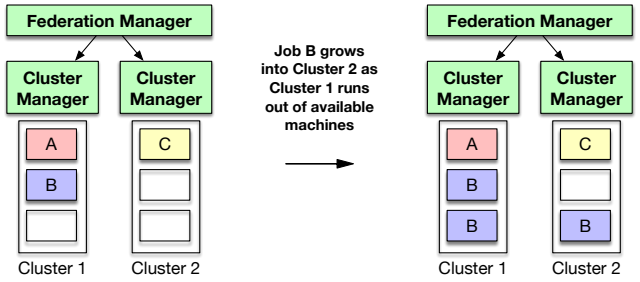
Separation of allocation and optimization responsibilities helps the allocator scale. The allocator makes quick decisions when starting tasks, whereas ReBalancer asynchronously runs a constraint solver to perform time-consuming global optimizations such as balancing CPU, network, and power.

Separation of responsibilities between Twine and application-level schedulers helps Twine scale further. Application-level schedulers handle many fine-grained resource allocation and lifecycle operations without involving Twine. For example, the Twine scheduler and allocator do not directly manage batch jobs, whose lifetime might last just a few seconds and cause high scheduling loads. The application-level batch scheduler acquires resources from Twine in the form of Twine tasks. It reuses these tasks over a long period of time to host different batch jobs, avoiding frequent host profile changes. The batch scheduler can create nested containers inside the tasks, similar to that in Mesos [17].

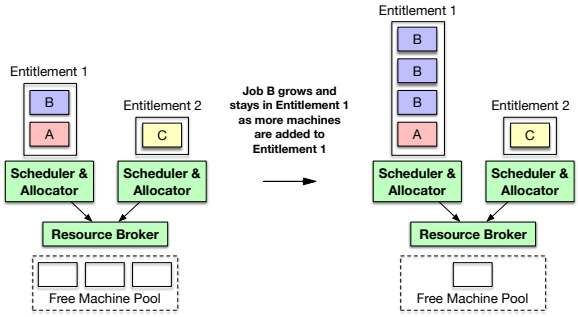
3.3 Comparison of Sharding and Federation

We acknowledge that Twine's scale of managing millions of machines is not unique, as Borg [39] and several public clouds likely manage infrastructure of that scale as well; however, we believe that Twine's approach is unique. Other cluster management systems scale out by deploying one isolated control plane per cluster and operate many siloed clusters. They pre-allocate machines to a cluster; once a job starts in a cluster, it stays with the cluster. This lack of mobility results in stranded capacity when some clusters are overloaded while others are idle. It also causes operational burden during cluster-wide maintenance such as hardware refresh, as shown in Figure 1.

To avoid stranded capacity, we can introduce mobility by moving either jobs or machines. To that end, the federation approach (e.g., Kubernetes Federation [25]) allows a job to be split across multiple static clusters, whereas Twine dynamically moves machines in and out of entitlements. Figure 13 compares these two approaches.



(a) Federation approach. This approach uses a Cluster Manager per cluster and introduces an additional Federation Manager layer. Each cluster has a set of statically configured machines. As job `B` in `Cluster 1` keeps growing, it overflows into `Cluster 2`.



(b) Twine's sharding approach. As job `B` grows, Twine adds more machines to `Entitlement 1`, and job `B` stays with the same entitlement and scheduler shard.

Figure 13: The two figures above contrast how federation and sharding support a job growing over time without stranding capacity in isolated clusters.

The federation approach can support complex multi-region, hybrid-cloud, or multi-cloud deployments, but it adds complexity as a scale-out solution. In order to provide a seamless user experience, the Federation Manager in Figure 13a has to perform complex coordination for a job whose metadata and management operations are split among multiple distributed Cluster Managers. In contrast, Twine is simpler for scaling out because a job is exclusively managed by one scheduler shard, and Resource Broker provides a simple interface to manage the shared regional pool of machines.

4 Availability and Reliability

Compared with the traditional approach of deploying one control plane per cluster, Twine's regional control plane incurs additional risks: 1) a control plane failure may impact all jobs in a region as opposed to just a cluster, and 2) network partitions may result in a regional Twine scheduler unable to manage an isolated DC.

Design principles. We observe several design principles to mitigate the risks listed above.

- **All components are sharded:** Each shard manages a small fraction of machines and jobs in a region, limiting the impact of a shard failure. Assuming Twine uses 20 scheduler shards to manage a 150MW region, each scheduler shard manages 7.5MW worth of machines, which is no bigger than a traditional cluster.
- **All components are replicated:** Consider schedulers for example: replicas of a scheduler shard sit in different DCs and elect a leader to process requests. If the leader fails or its network is partitioned from other DCs, a follower in another DC becomes the new leader.
- **Tasks keep running:** Even if all Twine components fail, existing tasks continue to run. New jobs cannot be created and existing tasks cannot be updated until Twine recovers. If a DC is partitioned from the scheduler, existing tasks in the DC continue to run.
- **Rate-limit destructive operations:** It is possible that a bug or fault might cause Twine to perform a large number of destructive operations quickly, e.g., shuffling tasks across machines at a fast pace. We protect against this failure by ensuring all components have fail-safe mechanisms to rate-limit destructive operations.
- **Network redundancy:** Fabric Aggregator connects our data centers in a region and can "*suffer many simultaneous failures without compromising the overall performance of the network* [14]." We did not experience within-region network partitioning as a major challenge.

Operational principles. In addition to the design principles listed above, we observe several operational principles.

- **Twine manages itself:** To avoid developing yet another cluster management tool to manage Twine installations, all Twine components, except for the agent, run as Twine jobs. We developed automation to bootstrap the Twine ecosystem starting from scratch. The Twine agent has no dependencies on other Twine components and our bootstrapping mechanism directly sends commands to agents to start other Twine components as Twine tasks.

- **Twine manages its dependencies:** As we built confidence in Twine's bootstrapping automation, we ran all systems that Twine depends on as normal Twine jobs, including ZooKeeper, Delos [4], Configerator [35], and a few other systems for storage, security, and continuous delivery. Twine managing itself and its dependencies improves reliability by eliminating the risk associated with maintaining specialized cluster management tools [8].
- **Gradual but frequent software release:** A new release progresses gradually across regions and shards so that a bug does not hit the entire fleet instantaneously. All components are released weekly or more frequently to lower the risk associated with large changesets.
- **Recurring large-scale failure test [38]:** This happens regularly in production to verify Twine's reliability.

These principles help us run Twine reliably. We share one anecdote where rate-limiting mitigated the risk caused by the complex interplay of four concurrent events: 1) shifting traffic from region X to region Y, 2) performing a load test in region Y, 3) adding new server racks to region Y before removing old racks, and 4) software upgrade for the web tier. The first three events led to increased power consumption in region Y and power capping on many machines. The scheduler rate-limited the number of tasks moving away from power-capped machines. This rate-limiting halted the web tier's software upgrade and protected against further loss of capacity. In this incident, rate-limiting provided a safety net before we debugged the incident.

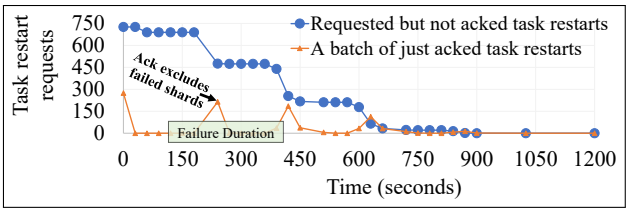
5 Evaluation

Our evaluation answers the following questions:

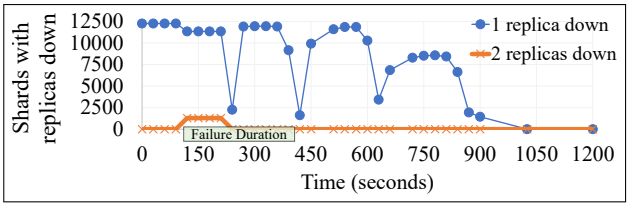
1. How does TaskControl deal with complex scenarios that impact an application's availability?
2. How effective is autoscaling for production use?
3. How effective are host profiles in improving performance? What is the overhead of switching host profiles?
4. How cost effective are small machines in replacing big machines?

5.1 TaskControl

Figure 14 demonstrates how TaskControl handles the complex situation of a software release and machine failures happening concurrently. This experiment uses a caching service managed by Shard Manager (§2.7). The cache's data are partitioned into 15,000 shards, and each shard runs three replicas. The 45,000 shard replicas are hosted by 1,000 Twine tasks. Shard Manager's TaskController helps minimize the risk of a shard losing more than one replica, i.e., driving Figure 14b's `2 replicas down` curve towards zero.



(a) Acknowledged and pending task-restart requests.



(b) Shards with some replicas down.

Figure 14: TaskControl helps a stateful service uphold its availability in the event of a concurrent software release and hardware failures.

Let `T`x denote the moment of x seconds into the experiment. At `T0`, the user initiates a rolling update of the service. In Figure 14a, at `T0`, the TaskController allows 274 tasks to update concurrently (the bottom curve). It does not allow any of the other 726 tasks to update (the top curve) because that would cause some shards to lose their second replicas. In Figure 14b, at `T0`, 12,264 shards lose one replica (the top curve) because they are hosted by the 274 tasks undergoing update. No shard loses its second replica (the bottom curve) because of the TaskController's precise shard availability calculation.

During the `Failure Duration` in the figures (between `T120` and `T415`), we inject the failure of one MSB that kills 50 tasks causing 1,292 shards to lose their second replicas, because those shards are also hosted by the 274 tasks undergoing update. The spike in Figure 14b's bottom curve reflects the impact on the 1,292 shards.

By `T240`, the 274 tasks are updated and become healthy. As a result, even if the 50 tasks in the failed MSB are still down, shards with `2 replicas down` drop to zero (the bottom curve in Figure 14b). At `T240`, the TaskController carefully selects the second batch of 214 tasks to update, ensuring no overlap between the shards hosted by the 214 tasks and the shards hosted by the 50 tasks in the failed MSB (see `Ack excludes failed shards` in Figure 14a). This careful task selection keeps Figure 14b's `2 replicas down` curve at zero throughout the rest of the experiment.

5.2 Autoscaling

Currently, we autoscale ≈800 services. Figure 15 shows the efficacy of autoscaling on our web tier, which is our largest service. Autoscaling frees up to 25% of the web tier's machines during off-peak hours. The bottom curve represents the web tier's CPU utilization. The middle curve represents the web tier's real job size, i.e., the number tasks in the job.

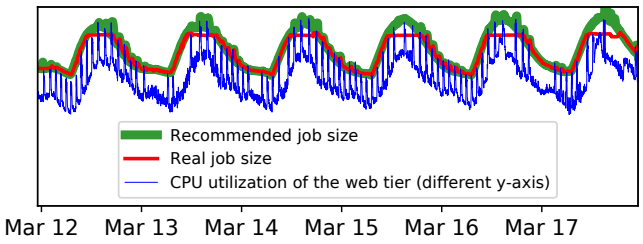


Figure 15: Autoscaling the web tier. The CPU spikes are caused by the continuous-delivery process restarting tasks.

The top curve represents autoscaling's recommendation for the job's ideal size. The CPU utilization closely follows the recommended job size, demonstrating the prediction's accuracy. Usually, the real job size also closely follows the recommended job size, but we intentionally choose a week when they diverged during peak hours.

During the week of March 12, 2020, our online products experienced a drastic traffic growth [19] related to COVID-19, causing a temporary capacity shortage. As a result, the real job size could not grow to follow the recommended job size during peak hours. The web tier's TaskController adapted to this unexpected situation without any manual intervention. During peak hours, it advanced the continuous-delivery software releases more slowly, bringing down fewer tasks concurrently to limit temporary capacity losses. During non-peak hours, it advanced software releases at a normal pace.

5.3 Host Profiles

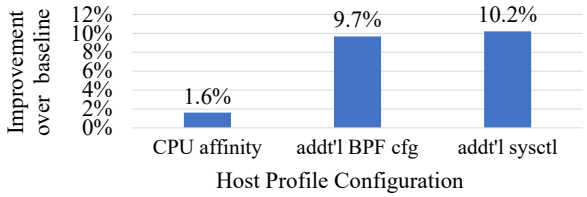


Figure 16: Host profiles improve the throughput of memcache.

Host profile's impact on application performance. We use memcache as an example to demonstrate how host profiles help improve application performance. We deploy a highly optimized version of memcache [30] on tens of thousands of machines. Figure 16 compares three host profiles versus the default settings. The baseline achieves 930K lookups per second on an 18-core/36-hyperthread machine. This extremely high throughput drives the need for host customization.

The `CPU affinity` host profile improves the throughput by dedicating 12 hyperthreads to handling NIC IRQs, one hyperthread to memcache's busy-loop thread, and 23 hyperthreads to memcache's worker threads. This separation avoids unnecessary interrupts and context switches. `Addl't BPF cfg` further reduces the overhead of certain BPF programs by lowering the packet sampling rate and disabling certain packet marking. `Addl't sysctl` further tunes 17 CPU scheduling and network settings, where improvements in reliability are more important than the mild performance gains. For example, based on lessons from past incidents, we tuned `net.ipv4.tcp_mem` to alleviate TCP's memory pressure under high loads in order to prevent cascading failures.

Overhead of switching host profiles. Figure 17 shows the host profile switching time. We discuss both ends of the performance spectrum. The P90 for enabling `CPU Turbo` takes 3.0 seconds. The P90 for enabling `HugePages` takes 244 seconds, as memory fragmentation sometimes causes the Linux kernel to fail to allocate hugepages and a machine reboot may be needed to finish the operation. To alleviate the problem, we recently developed a kernel improvement [33] that achieves above 95% success rate for hugepage allocation; we are still in the process of deploying it to production.

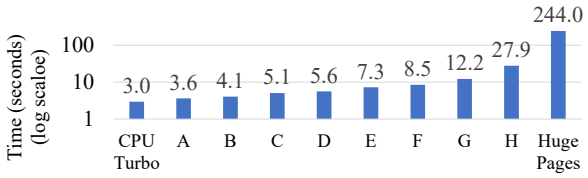


Figure 17: P90 host profile switching time for different host profiles.

On average, a machine changes its host profile once every two days; hence the overall overhead is negligible. Figure 18 depicts how autoscaling impacts host profile changes.

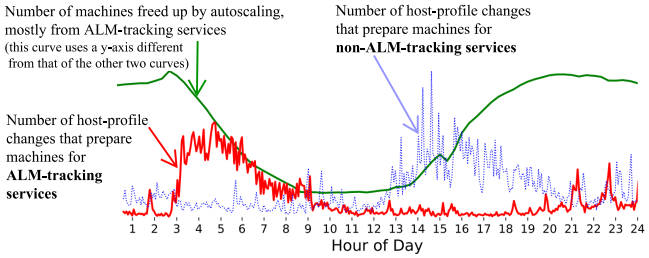


Figure 18: Autoscaling is the biggest driver for host profile changes. The load of an active last minute (ALM) tracking service is proportional to the number of people using our online products. In response to our products' changing load, Twine moves machines to entitlements running ALM-tracking services during hour 3 to 8 and to entitlements running non-ALM-tracking services during hour 13 to 20, respectively.

5.4 Power-efficient Small Machines

The total cost of ownership (TCO) of a machine includes the hardware cost, power consumption, and operating expense. We compare the TCO of small machines vs. big machines using the following metrics:

- *B*: The TCO of a big machine (dual CPUs and 256GB RAM) is *B* times that of a small machine (one CPU and 64GB RAM).

- S: A service needs S number of small machines to replace a big machine and achieve the same performance.
- $\frac{S}{B}$: Relative TCO (RTCO) of a service running on small machines vs. on big machines.

Figure 19 shows the RTCO of 22 fleet-wide representative services. One service has worse than 100% RTCO, seven use the maximum prescribed 100% RTCO, and a majority of services are able to achieve a better RTCO.

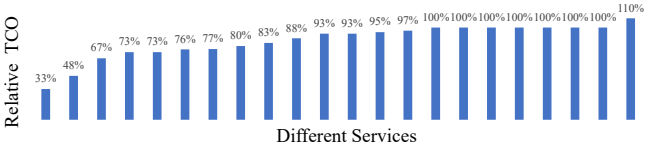


Figure 19: The relative total cost of ownership of services running on small machines vs. on big machines. Smaller numbers mean bigger savings.

The first service in Figure 19 achieves a low 33% RTCO by adopting Shard Manager (§2.7). The service is sharded; its biggest shard has 20x higher load than its smallest shard and the load varies. The service's previous static-sharding solution did not work well, whereas Shard Manager is able to balance the load via shard migration. After switching to small machines, the service better utilizes the overall higher CPU count of small machines under the same TCO.

The second service achieves a 48% RTCO by moving from an in-memory data store to an external flash-based database. Its 48% RTCO includes the cost of the database, which is only a small part of the total TCO.

The service with 76% RTCO is TAO [7], our social graph cache. CacheLib [5] provides transparent tiered memory on top of 64GB RAM and TBs of local flash to replace 256GB RAM (§2.8). Its 76% RTCO includes the cost of flash.

One outlier service has 110% RTCO, meaning it costs 10% more to run on small machines. The memory is used to store certain data indices and ML models that rank the indices. We are improving the service to target 90% RTCO, e.g., by leveraging CacheLib [5] to provide tiered memory.

Across all services in our fleet beyond the examples in Figure 19, we achieved an average 83% RTCO, i.e., 17% fleet-wide TCO savings. This also includes 18% power savings. Overall, we have been successful at using small machines.

6 Experience with Shared Infrastructure

As described in §1, Twine has allowed us to grow *twshared*, our shared compute pool, from ≈15% in 2019 to ≈56% in 2020. We share our experience with growing twshared.

6.1 Economies of Scale in twshared

Shared infrastructure provides economies of scale by reducing hardware, development, and operational costs. Examples:

- **Capacity buffer consolidation.** As services migrated into twshared, we consolidated siloed buffers for software releases, maintenance, fault tolerance, and growth into centralized buffers, improving utilization by ≈3%.
- **Turbo Boost.** We aggressively enabled Turbo on processor cores and relied on ReBalancer to mitigate power hotspots, improving utilization by ≈2% in 2020.
- **Autoscaling.** Autoscaling freed up over-provisioned capacity, reclaiming ≈2% of capacity in 2020.

As shown in Figure 20, as of October 2020, twshared's average memory and CPU utilization are ≈40% and ≈30%, respectively. For comparison, the figure also shows utilization for *private pools*, our legacy pools of customized machines dedicated to individual workloads. We plan to improve utilization through multiple approaches, such as the one described below. Our fleet is dominated by user-facing services that provision capacity for peak load. Autoscaling frees some of this over-provisioned capacity during off-peak hours and provides it as opportunistic capacity for other workloads to use. Unfortunately, we do not yet provide service-level objectives (SLOs) on the availability of opportunistic capacity, which is limiting adoption and usage of all available capacity. As we establish SLOs for opportunistic capacity, improve stacking, and consolidate capacity buffers, we expect twshared's utilization to increase.

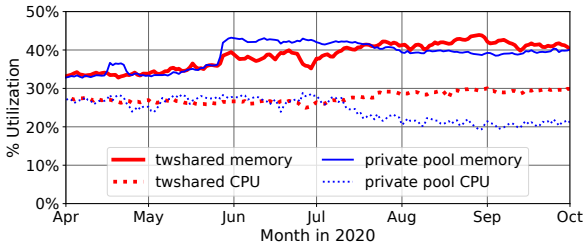


Figure 20: Daily average CPU and memory utilization of twshared and private pools circa October 2020.

6.2 Path to Shared Infrastructure

We had broad conversations with colleagues in industry and learned that while partial consolidation of workloads is common, no large company has achieved near 100% shared infrastructure consolidation. Further, we learned that cultural challenges are as significant as technical challenges. Below, we describe our strategy and major milestones towards migrating *all* non-storage workloads into twshared.

Make Twine capable of supporting a large shared pool. Scalability, entitlements, host profiles, and TaskControl are Twine's important features that enabled workload consolidation. The flexibility offered by host profiles and TaskControl ensures that twshared can support both 1) the general needs of thousands of services, and 2) the specialized needs of a smaller set of services that consume the majority of capacity.

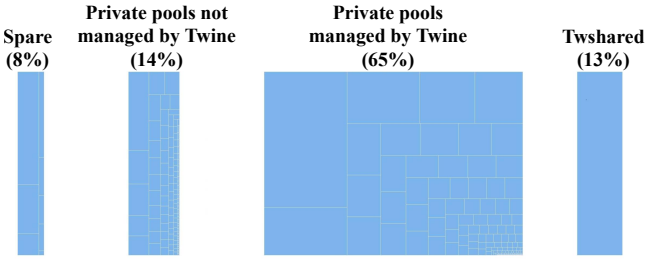


Figure 21: Breakdown of machines in our fleet as of **August 2018**. Each small rectangle inside a category represents a private pool, and its size is proportional to the number of machines in the private pool. There were hundreds of private pools, many of which were small in size. The percentages at the top reflect the number of machines in each category relative to all machines globally. From August 2018 to **October 2020**, the breakdown evolved from [8%, 14%, 65%, 13%] to [13%, 5%, 26%, 56%], where the numbers match the left-to-right categories in the figure.

Publicize the growth and health of twshared. We developed a tool to show the realtime breakdown of our fleet and the growth of twshared. A snapshot is shown in Figure 21. We consolidated the fragmented mechanisms of measuring machine health into the Health Check Service. Continuous improvements have resulted in twshared running healthier than private pools, 99.6% vs. 98.3%.

Set a strong example for others to follow. Early on, we targeted the web tier, our largest private pool. It directly serves external users of our company's products and any outage would be immediately noticeable. We finished migrating the web tier into twshared mid-2019. As the web tier team is highly respected in the company, their testimony motivated others to follow.

Make migration mandatory. After the web tier migration, we gained company-wide support for mandatory migration. Further, we established that all new compute capacity will land only in twshared. This mandate, along with Twine's flexibility of supporting customization through TaskControl and host profiles, has made twshared our ubiquitous compute pool.

6.3 Case Study of twshared Migration

PG_x is a large product group that runs hundreds of diverse services on hundreds of thousands of machines. Their services vary in size from a few machines to tens of thousands, and in complexity from computationally intensive ML training to latency-sensitive ad delivery. Previously, their fleet was fragmented into tens of private pools per region. The first PG_x service migrated into twshared in January 2020; as of September 2020, more than 70% of PG_x machines run in twshared. Given the size and diversity of their services, we expect the migration to finish in late 2021.

PG_x services use hundreds of twshared entitlements; if a service runs in multiple regions, it needs one entitlement per region. Figure 22 shows the size distribution with the biggest entitlement running ≈2K jobs on ≈15K machines.

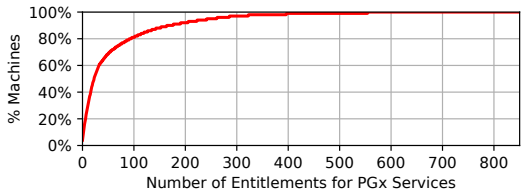


Figure 22: CDF of PG_x entitlement size. The distribution is highly skewed. The largest 54 entitlements account for 70% of PG_x capacity in twshared.

Accommodating workload-specific requirements helps onboard PG_x services onto twshared. For instance, many PG_x services run A/B tests in production, e.g., to evaluate the effectiveness of a new model–these services need to explicitly configure the processor generation for their tasks to prevent performance variations between hardware types from polluting their test results.

The capacity guaranteed by entitlements and private pools account for 55% of PG_x machines. The remaining 45% are from opportunistic sources including capacity buffers, machines freed up by autoscaling, and unused portions of other teams' entitlements. *Optimus* is an application-level scheduler that runs atop Twine to manage opportunistic capacity. When opportunistic capacity is not available, some services gracefully degrade their quality of service.

Jobs with a TaskController consume 36% of PG_x capacity in twshared; in total these jobs use three different TaskControllers, including the one from Shard Manager [16]. About 95% of PG_x capacity is consumed by entitlements that use some combination of these three host profile settings:

1. If a service does frequent flash writes, it prefers the flash drive to expose only a fraction of the flash capacity in order to reduce write amplification and burn rate.
2. If a service can fully utilize a whole machine and does not stack with other services, we disable the cgroup2 CPU controller to eliminate its overhead.
3. Because our data centers are power constrained and CPU Turbo consumes extra power, we enable Turbo only for services that can benefit significantly from Turbo and are running in selected data centers with sufficient power.

Overall, our experience with PG_x indicates that, despite the significant upfront effort needed for migration, even large and varied services are motivated to adopt shared infrastructure that reduces their operational burden. PG_x' success in using opportunistic capacity at a large scale has spurred us to develop SLO guarantees and drive broader adoption (§6.1). Entitlements, TaskControl, and host profiles enable customization in a shared pool and were the features that enabled the migration. On the other hand, PG_x services have grown to hundreds of entitlements within 9 months, motivating us to address entitlement fragmentation (§7.1).

7 Lessons Learned

Evolving Twine and growing twshared has taught us several lessons. We share some highlights and lowlights below.

7.1 Entitlement Fragmentation

We overloaded entitlements with two responsibilities: fleet partitioning and quota management. Entitlements partition millions of machines into smaller units that can be effectively managed by scheduler shards. Twine jobs can only stack within the same entitlement, implying that an entitlement be sized at a few thousand machines, similar to a Borg [39] cell.

On the other hand, leveraging entitlements for quota management results in small entitlements. For example, an important service may wish for an entitlement with 10 tasks rather than a larger entitlement shared with other services to protect against the risk that a rogue service grows unexpectedly and uses up the entitlement quota.

We are in the process of splitting an entitlement's responsibility into two new abstractions: a *materialization* for fleet partitioning and a stackable *reservation* for quota management. A *materialization* functions as a pseudo cluster, has a host profile associated with it, and is always large enough to enable job stacking across thousands of machines.

7.2 Controlled Customization

Our goal is ubiquitous shared infrastructure. A difficult lesson we learned from the first six years of operating twshared was that customization is key to migrating services over. For instance, without host profiles, our web tier and memcache services would not run in twshared as their performance would regress by 11% and 10.2% respectively. TaskControl has provided a path for stateful services such as TAO [7] and MySQL to deprecate their custom cluster management tooling and adopt Twine and shared infrastructure.

We prioritize maintainability when deciding what customization to permit. Currently, we offer 17 host profiles and 16 TaskControllers to support thousands of services. Our recent migration of ≈70% of a large product group's services into twshared (§6.3) leveraged existing host profiles and TaskControllers.

In hindsight, we permitted some customizations that appeared useful initially, but later became barriers for fleet-wide optimizations. For example, a job's tasks are identical by default, but we provided the ability to customize individual tasks, including the executables to run, command line options, environment variables, and restart policies. Developers abused this customization to implement simple sharding so that each task does different work. Autoscaling changes the number of tasks in a job and breaks the job's task customization. As we enable autoscaling for all ALM-tracking services, we are removing task customization and migrating these services to use Shard Manager [16] instead.

7.3 Supporting Global Services

Many developers wish to run a global service without worrying about operational challenges: which regions to deploy to, how much capacity is needed in each region, and how to handle regional failures. We currently operate multiple global Twine deployments that spread a global job's tasks across regions, similar to how a regional Twine deployment spreads a regional job's tasks across data centers in a region. Currently, global jobs account for 8% of all our jobs.

We have learned over time that global Twine deployments did not provide the right abstraction for managing global services. Machines in a region are largely fungible due to the high network bandwidth and low latency within a region, but this is not true for machines distributed across regions. Hence, it is better to explicitly decompose a service's global capacity needs into capacity needs for specific regions, as opposed to global allocators making ad hoc decisions on which regions to get machines from. We are replacing global Twine deployments with a new Federation system built atop regional Twine deployments to provide stronger capacity guarantees and more holistic support for a global-service abstraction.

7.4 Challenges with Small Machines

Our decision to leverage small machines brings with it numerous trade-offs. The effort to rearchitect and reimplement memory-capacity-bound services was higher than we anticipated. On the other hand, we leveraged this opportunity to holistically modernize our legacy services, e.g., moving from static sharding to dynamic sharding for better load balancing. As small machines run contrary to the industry practice of favoring big machines; we need to work closely with hardware vendors to optimize machines for our internal workloads, e.g., removing unneeded NUMA components.

That said, the 18% power efficiency win (§5.4) from small machines has been worth the above trade-offs. We intend to continue using small machines in the coming years, but are also prepared to evolve our hardware strategy as needed. Two factors lead to our decision of adopting small machines: 1) our legacy large services were optimized for utilizing entire machines running in private pools, and 2) our stacking technology needed to mature and improve support for performance isolation [42]. As our services undergo architectural changes to run effectively in twshared, and we improve our stacking technology, we may revisit our hardware strategy.

8 Related Work

Scalability and scheduling performance. Kubernetes [25] and Hydra [9] scale out through federation, whereas Twine scales out through sharding. Figures 13 compares the two approaches. A large body of work [6, 15, 20, 31] focuses on improving batch scheduling throughput and latency. Twine

delegates the handling of short-lived batch jobs to application-level batch schedulers. This separation of concerns helps Twine scale, as discussed in §3.2.

Entitlements. Twine has some similarity to the two-level schedulers (Mesos [17], YARN [37], Apollo [6], and Fuxi [44]), with Twine entitlements as resource offers and Twine scheduler shards as Application Masters (or *frameworks* in Mesos). However, the bottom-level Resource Manager (or *Master* in Mesos) is designed for the scale of a single cluster. In contrast to the single-master two-level architecture, we propose a three-level architecture with sharding so our design scales out: Resource Broker manages machines, Twine scheduler manages containers, and Application-level schedulers manage workloads such as batch and ML.

Kubernetes' cluster autoscaler [24] can respond to workload growth by provisioning VMs in a public cloud and adding them to a node pool. Kubernetes' resizable node pool corresponds to Twine's entitlement, and a public cloud's available resources correspond to Twine's shared free machine pool maintained by Resource Broker. Decoupling Kubernetes and cloud makes the setup flexible, but also misses optimization opportunities compared with Twine's integrated ecosystem. Multiple Kubernetes clusters run independently without coordination, whereas Twine's ReBalancer performs global optimization across entitlements, and an entitlement can be migrated across scheduler shards.

TaskControl. The two-level schedulers (Mesos [17], YARN [37], Apollo [6], and Fuxi [44]) allow their applications to provide custom Application Masters. The interface with Application Masters is for negotiating resource allocation, e.g., "*requesting N containers with X CPU and Y memory*," whereas the TaskControl API is for negotiating lifecycle management, e.g., "*delaying restarting task T*."

Kubernetes [23]'s custom controllers provide a universal extension framework that can be used to implement various custom functions like autoscaling and injecting sidecars for traffic routing. In contrast, TaskControl exclusively focuses on allowing or delaying task lifecycle operations. This narrow interface strikes a balance between standardization and customization (§7.2) and prevents proliferation of customizing all aspects of the Twine control plane. We are unaware of any Kubernetes custom controller that specifically offers extension points to allow or delay task lifecycle operations.

Azure supports update domains and fault domains [3] and the example stateful service in Figure 2 can improve availability by spreading its data shards' replicas across those domains. However, in the event of a machine failure, Azure may still proceed with a rolling update that can lead to unavailable shards because it does not know precisely how the shard replicas are spread across fault domains and update domains.

Host profiles. Paragon [12] schedules a job on machines that are beneficial to the job's performance, but it does not reconfigure a machine.

Some systems statically partition machines in a cluster and preconfigure their hardware and OS settings to suit different workloads. Others dynamically adjust predetermined settings (e.g., Turbo [40]) based on runtime profiling, while disallowing other customizations (e.g., btrfs vs. ext4). We believe that Twine is the first system that 1) allows workloads to provide customized hardware and OS settings to run in a shared machine pool and 2) dynamically reconfigures a machine just-in-time as the workload is scheduled onto the machine. On average, Twine reconfigures a machine once every two days, primarily due to Autoscaling (see Figure 18).

Power-efficient hardware. A large body of work studies power-efficient computing [1, 10, 27]. Our infrastructure is unique in 1) using power-efficient small machines as a universal computing platform, and 2) consolidating towards a single compute machine type (one CPU and 64GB RAM), as opposed to offering a variety of high-memory or high-CPU machine types. Both approaches required our workloads to make software architectural changes that would be challenging in a public cloud with external customer workloads.

Overcommitment and autoscaling. Past work overcommits CPU and memory by colocating batch jobs and online services [11, 22, 39, 43]. Twine does not overcommit CPU or memory by default, although a job owner can explicitly configure their job to do so. On the other hand, we overcommit power by default [41], as power is our most constrained resource. Twine helps mitigate power hotspots by relocating tasks across data centers. Twine's SRM uses historical data to predictably adjust the number of tasks in a job. Borg's Autopilot [34] adjusts the CPU and memory allocated to each task–this is an area of future work for Twine.

9 Conclusion

We identify existing cluster management systems' limitations in supporting large-scale shared infrastructure. We describe our novel solution that allowed us to scale Twine to manage one million machines in a region, move jobs across physical clusters, collaborate with applications to manage their lifecycle, support host customization in a shared pool, use power-efficient small machines to achieve higher performance per watt, and employ autoscaling to improve machine utilization. We share our experience with twshared and our strategy towards ubiquitous shared infrastructure.

Acknowledgments

This paper presents the engineering work of several teams at Facebook that have built Twine and its ecosystem over the past decade. We thank Niket Agarwal, Marius Eriksen, Tianyin Xu, Murray Stokely, Seth Hettich, and the OSDI reviewers for their insightful feedback.

References

[1] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[2] Alexey Andreyev. Introducing data center fabric, the next-generation Facebook data center network, 2014. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[3] Azure update domain and fault domain, 2019. https://docs.microsoft.com/en-us/azure/virtual-machines/availability.

[4] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[5] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[6] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[7] 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*, 2013.

[8] Christopher Bunn. Containerizing ZooKeeper with Twine: Powering container orchestration from within, 2020. Facebook blog post. https://engineering.fb.com/developer-tools/zookeeper-twine/.

[9] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[10] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-Intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017.

[12] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[13] Jeffrey Dunn. Introducing FBLearner Flow: Facebook's AI backbone, 2016. https://engineering.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/.

[14] João Ferreira, Naader Hasani, Sreedhevi Sankar, Jimmy Williams, and Nina Schiff. Fabric Aggregator: A flexible solution to our traffic demand, 2014. Facebook blog post. https://engineering.fb.com/data-center-engineering/fabric-aggregator-a-flexible-solution-to-our-traffic-demand/.

[15] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[16] Gerald Guo and Thawan Kooburat. Scaling services with Shard Manager, 2020. Facebook blog post. https://engineering.fb.com/production-engineering/scaling-services-with-shard-manager/.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott

Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center . In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[18] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[19] Mike Isaac and Sheera Frenkel. Facebook Is 'Just Trying to Keep the Lights On' as Traffic Soars in Pandemic. *The New York Times*, 2020. https://www.nytimes.com/2020/03/24/technology/virus-facebook-usage-traffic.html.

[20] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[21] Rui Jian and Hao Lin. Tangram: Distributed Scheduling Framework for Apache Spark at Facebook, 2019. https://databricks.com/session/tangram-distributed-scheduling-framework-for-apache-spark-at-facebook.

[22] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[23] Kubernetes, 2020. https://kubernetes.io/.

[24] Kubernetes cluster autoscaler, 2020. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[25] Kubernetes Federation, 2020. https://github.com/kubernetes/community/tree/master/sig-multicluster.

[26] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sree Kodakara, David Lo, and Partha Ranganathan. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[27] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V Vasilakos. Cloud Computing: Survey on Energy Efficiency. *Acm computing surveys (csur)*, 47(2):1–36, 2014.

[28] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. Turbine: Facebook's Service Management Platformfor Stream Processing. In *Proceedings of the 36th IEEE International Conference on Data Engineering*, 2020.

[29] Aravind Narayanan, Elisa Shibley, and Mayank Pundir. Fault tolerance through optimal workload placement, 2020. Facebook blog post. https://engineering.fb.com/data-center-engineering/fault-tolerance-through-optimal-workload-placement/.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[31] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[32] Vijay Rao and Edwin Smith. Facebook's new server design delivers on performance without sucking up power, 2016. https://engineering.fb.com/data-center-engineering/facebook-s-new-front-end-server-design-delivers-on-performance-without-sucking-up-power/.

[33] Roman Gushchin. Hugetlb: optionally allocate gigantic hugepages using cma, 2020. https://lkml.org/lkml/2020/3/9/1135.

[34] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at Google. In *Proceedings of the 15th ACM European Conference on Computer Systems*, 2020.

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

[36] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of the 15th ACM European Conference on Computer Systems*, 2020.

[37] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[38] 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.

[39] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[40] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[41] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's Data Center-Wide Power Management System. *ACM SIGARCH Computer Architecture News*, 44(3), 2016.

[42] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[43] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[44] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, 2014.