Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta

Boris Grubic, *Meta;* Yang Wang, *Meta and the Ohio State University;* Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, and Dan Kelley, *Meta;* Soteris Demetriou, *Meta and Imperial College London;* Kenny Yu and Chunqiang Tang, *Meta*

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

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

Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta

Boris Grubic[1], Yang Wang[1,2], Tyler Petrochko[1], Ran Yaniv[1], Brad Jones[1], David Callies[1], Matt Clarke-Lauer[1], Dan Kelley[1], Soteris Demetriou[1,3], Kenny Yu[1], and Chunqiang Tang[1]

[1]Meta Platforms, [2]The Ohio State University, [3]Imperial College London

Abstract

We present *Conveyor*, Meta's software deployment tool, along with the valuable data obtained from managing over 30,000 deployment pipelines that deploy all kinds of services at Meta across millions of machines. We describe a wide range of deployment scenarios that Conveyor supports to achieve universal coverage. At Meta, out of all the deployment pipelines for services deployed via containers, 97% of them employ fully automated deployments without manual intervention: 55% utilize continuous deployment, instantly deploying every code change to production after passing automated tests, and the remaining 42% are automatically deployed on a fixed schedule (mostly daily or weekly) without manual validation. We highlight several distinguishing features of Conveyor, including safe in-place updates to reduce hardware costs, analysis of code dependencies to prevent faulty releases, and the capability to deploy complex ML models at scale.

1 Introduction

"*Release early, release often*" [1, 32] is central to Meta's engineering culture. For example, Meta's largest service, *FrontFaaS*, which is a serverless function-as-a-service platform, runs on more than half a million machines and has tens of thousands of developers making changes to its code base, with thousands of code commits every workday. Despite this extremely dynamic environment, it continuously releases a new version into production every three hours [33].

Although the concept of frequent software releases is well-established, previous studies have primarily relied on limited surveys or analyses [7, 20, 23, 25, 38–41, 48]. In contrast, we leverage our nine years of direct experience in developing Meta's deployment tool called *Conveyor* and the wealth of data obtained from managing over 30,000 deployment pipelines to answer the following questions: 1) What is the adoption rate of deployment automation, and what is important in driving the adoption? 2) What is special about deployment safety at hyperscale? 3) What distinguishes the deployment of ML models from traditional service executables? We summarize our answers to these questions below.

1.1 Adoption of Deployment Automation

Universal adoption. We strongly argue for universal adoption of a *single* deployment tool within an organization to support all kinds of services, both small and large. At Meta, 0.1% and 1% of the largest services consume 40% and 80% of the total fleet capacity, respectively. Similarly, Google reported that "*the top 1% of jobs consume over 99% of all resources* [47]." These largest services often require the most complex deployment features, and neglecting them would lead to fragmentation and difficulties in managing the site. For example, due to FrontFaaS's demanding requirements, it used to have its own complex deployment tool written in over 30,000 lines of code. This kind of fragmentation complicates the operation of our site. In the event of a site outage, very few people know how to safely revert a specific service's problematic release.

Furthermore, the impact of a deployment tool on site reliability necessitates many advanced features to ensure the safety of deployments, as outlined in §1.2. While it may be tempting to develop a new custom tool to address specific needs that are currently unsupported by the standard tool, Meta's experience has consistently shown that these custom tools, owned by individual product teams, have seldom reached the level of maturity required to provide the essential, advanced deployment-safety features. Consequently, without any exceptions, these custom tools have always been assimilated back into the standard tool as it evolves and matures.

Over the past nine years, Conveyor has achieved universal adoption at Meta, and the key to its success lies in its ability to support a wide range of deployment scenarios while ensuring the safety of deployments (§3).

Fully automated deployments. After addressing numerous challenges along the way, the adoption rate of *continuous deployment* [39] at Meta has greatly exceeded our initial expectations. With continuous deployment, every time a code change is committed to the code repository, it automatically goes through a series of tests. If it passes those tests, it is deployed to production immediately, without manual intervention. Currently, out of all Meta's deployment pipelines for services deployed via containers, 97% employ fully automated deployments: 55% utilize continuous deployment and the remaining 42% automatically deploy on a fixed schedule (mostly daily or weekly) without manual validation.

In contrast, in early 2018, 47% of services at Meta were deployed manually without using any automation tool, 41% utilized an automation tool but still required manual validation, and only 12% were deployed through full automation. The significant increase in fully automated deployments, from 12% to 97%, has greatly reduced developer toil and improved productivity. Moreover, in early 2017, 20% of our fleet's RPC traffic [37] were generated by executables that had not been updated within 30 days. Currently, this number has dropped to around 1%. Up-to-date code brings many benefits, such as faster iteration speed and the timely implementation of bug fixes and security fixes.

To automate deployments, bugs and deployment failures must be embraced as a norm. Instead of introducing manual validations to prevent failures, automated guardrails such as testing and health checks should be implemented to detect release failures early and contain their adverse effects. Specifically, although 5.4% of our deployments fail, deployments are still highly reliable as the majority of those failed deployments are caught during early deployment phases with little to no production impact. The high reliability of automated deployments is evidenced by the fact that only 0.92% of finished deployments are manually reverted or patched by developers.

1.2 Deployment Safety at Hyperscale

Making in-place updates safe. Due to its strong safety guarantees, the approach of *mirroring update*, which keeps the existing deployment intact and uses a separate set of containers to deploy a new version of the software, is widely used in the industry [2, 11], as it can easily redirect traffic back to the old deployment if the new deployment encounters issues. However, the cost of keeping spare hardware for a second deployment is prohibitive for our hyperscale services. At Meta, we exclusively utilize *in-place updates* for all services, which directly update containers in the existing deployment, eliminating the need for a separate deployment.

Updating a service in place requires precise controls over container updates and health checks to ensure safety. To enable such controls, we have enhanced our cluster manager [45] to allow updating a specific subset of a job's containers to a new version while keeping the remaining containers on the old version, as opposed to the traditional approach that requires updating a job's all containers as a whole [22].

Furthermore, for complex services like sharded databases, it is essential for the service itself, rather than the cluster manager, to determine when to update each container, because the service knows best about its own requirements such as shard replica safety. Our cluster manager's TaskControl interface enables this, whereas existing cluster managers disallow it.

Finally, we have enhanced our monitoring system to conduct health checks on "*moving targets*," i.e., a dynamically changing subset of a job's tasks. This subset evolves as the deployment progresses, in contrast to traditional health checks that always target a fixed set of tasks, i.e., all tasks in the job. Overall, these precise-control features enable Conveyor to perform in-place updates safely while eliminating the extra hardware costs associated with the mirroring approach.

Handling complex code dependencies. Pioneered by hyperscalers such as Google, monorepo [9], which stores the code for an organization's many projects in a single repository, has become increasingly popular due to benefits such as improved code reuse. However, increased code reuse leads to more complex code dependencies, such as a service X transitively depending on shared code at a depth of over 10 layers. In such cases, when a bug is introduced to some dependent code, the owner of service X might not even know that service X is affected. Our data show that about 14% of the to-be-deployed executables are affected by known bugs in dependent code and should not be deployed into production. Conveyor's Bad Package Detector automatically enforces this (§3.2), but existing deployment tools do not support it.

1.3 ML Model Deployment

Traditional deployment tools [3, 18, 42, 46] exclusively focus on the deployment of service executables. Even if they evolve to achieve universal coverage for service executables, in the era of rapid proliferation of ML applications, they still leave a significant gap by not addressing the deployment of ML models. Conveyor has been specifically enhanced to address this need and currently about 44% of its pipelines are for model deployments. To support ML models, Conveyor coordinates deployment pipelines for models that share the same inference executable, synchronizes the deployment of different shards of a partitioned large model, and implements phased in-place updates of models through the configuration management system [44], in contrast to the traditional approach of updating executables via the cluster manager. Dedicated ML platforms such as AWS SageMaker can deploy models using the mirroring approach [35, 36], but they do not support the advanced model-deployment features mentioned above.

Contributions. We make several contributions in this paper.

- We believe this paper is the most comprehensive report to date on deployment scenarios, operational experience, and production data related to software deployment.
- We demonstrate the feasibility of achieving aggressive goals for software deployment, such as frequent and automated deployment without manual validation, and using a single deployment tool to provide universal coverage for datacenter services, ML models, application configurations, host-level daemons, and mobile apps.
- We present novel techniques that ensure the safety of in-place updates, prevent faulty releases through analysis of code dependencies, and safely deploy ML models.

2 Overview of Software Deployment at Meta

To provide context for the discussions in later sections, this section gives an overview of Meta's software deployment culture and deployment ecosystem.

2.1 Deployment Culture

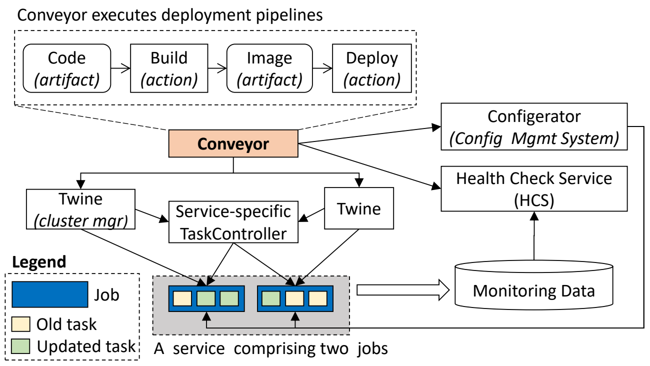
At Meta, we mandate frequent software deployments for multiple reasons. First, frequent releases enhance developer productivity. Engineers at Meta heavily rely on A/B test results of new product features to guide their development. This necessitates frequently deploying and testing new code with real users. Second, frequent releases reduce the complexity of troubleshooting in production because each release contains fewer code changes. Finally, frequent releases ensure timely deployment of bug fixes and security fixes. Consider a widely publicized site outage in 2014 [29]. Two months before the outage, the issue that later caused the outage was identified and the bug fix was already committed to the code repository. Unfortunately, no new deployment took place for two months. In general, preventing human mistakes in software deployments at the scale of thousands of engineers is unachievable without utilizing automation tools like Conveyor.

Meta's Push4Push program enforces regular deployments through tickets. Service owners get a ticket if their services are not updated within certain days (42 days for low-traffic services and 30 days for high-traffic services), and it escalates to managers at 63 days. In practice, Push4Push results in 96% of services deploying weekly or more frequently.

2.2 Deployment Ecosystem

Meta's private cloud comprises multiple datacenter regions. Each region has its own instance of cluster manager called Twine [45], which manages machines and containers. During a deployment, Conveyor instructs Twine to update containers. Meta's datacenter software is structured as many microservices [21]. A service comprises one or multiple *jobs*. A job comprises one or more *tasks*, and a task is mapped to a Linux container. Typically, a service is deployed to multiple regions for resilience, running one job for each region where it is deployed, and each of those jobs is managed by a different Twine instance.

Figure 1 presents an overview of the software deployment ecosystem at Meta. Developers define the update procedure for their services by specifying a deployment *pipeline*, which is a directed acyclic graph (DAG) comprising a set of input *artifacts* and a set of *actions*. A simple pipeline is shown at the top of Figure 1, while a more complex example is provided in Figure 3. An *action* represents an operation to be executed and takes a set of *artifacts* as input and potentially generates a new set of artifacts. Examples of artifacts include source code and compiled executables. Once all the input artifacts of a pipeline are ready, the pipeline will be executed, creating a *release*. A release is one execution of a pipeline.



Conveyor executes deployment pipelines
Code (artifact)
Build (action)
Image (artifact)
Deploy (action)
Conveyor
Configerator (Config Mgmt System)
Health Check Service (HCS)
Twine (cluster mgr)
Service-specific TaskController
Twine
Monitoring Data
Legend
Job
Old task
Updated task
A service comprising two jobs

Figure 1: Software deployment ecosystem at Meta. The two Twine instances manage jobs in different datacenter regions.

The `deploy` action drives Twine to update containers and is typically the most complex part of a pipeline. To prevent a bug from instantly impacting all tasks within a job, the `deploy` action updates tasks in *phases*. Each phase updates a subset of tasks, checks their health, and proceeds to the next phase only if no issues are detected. To check service health, Twine and the service itself can log various health signals, such as CPU utilization and user engagement metrics. The Health Check Service (HCS) is responsible for checking these health signals for anomalies based on user-defined rules. For example, it can detect if user engagement drops below a certain threshold. Collecting health signals often requires a waiting period to gather monitoring data, known as the `bake` time. Therefore, a `deploy` action typically includes several phases, each with an update period and a bake period, and such information is defined in a *deployment plan*.

In addition to driving Twine to deploy service executables, Conveyor can also drive Configerator [44], our configuration management system, to implement phased deployments of ML models and configuration files (§3.3). Moreover, a service with special requirements can optionally provide its own TaskController to advise Twine on which tasks are safe to update together, as explained in the example below.

2.3 Component Interaction by Example

We illustrate the interaction between various components in Figure 1 through the example of deploying a new software version for a sharded key-value store (KVStore). The KVStore is deployed across two regions, denoted as X and Y, with each region running a separate job consisting of six tasks. These jobs and tasks are labeled as $JobX = [X1, \cdots, X6]$ and $JobY = [Y1, \cdots, Y6]$. These 12 tasks collectively host 500 data shards, each of which has three replicas that are potentially distributed across regions.

According to the KVStore's quorum protocol, if a shard loses two out of its three replicas, it becomes unavailable. Thus, concurrently updating any two specific tasks carries the risk of rendering certain shards unavailable. Since Conveyor and Twine are unaware of the application-level shard

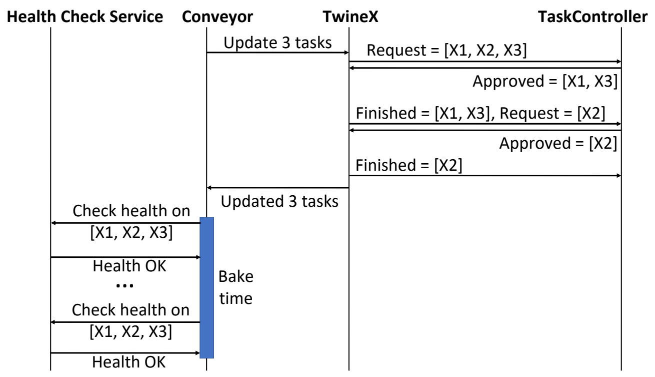


Figure 2: Phase 1 of updating the KVStore.

placement [24], they cannot determine whether it is safe to update any two specific tasks together. To ensure shard availability, the KVStore provides its own custom TaskController to advise Twine on which tasks are safe to update together.

The `deploy` action of the KVStore consists of two phases: 1) updating three tasks of *JobX*, and 2) updating the remaining three tasks of *JobX* and all six tasks of *JobY* (see §6.5.2 for alternative setups). In phase 1, Conveyor instructs *TwineX*, the Twine instance in region X, to update three tasks of *JobX*, as depicted in Figure 2. *TwineX* chooses to update tasks $X1$, $X2$, and $X3$ and communicates this intent to the TaskController by sending $request = [X1, X2, X3]$. The TaskController responds with a subset of tasks that can be safely updated together. For example, the TaskController may find that the shard replicas hosted by $X1$ and $X3$ do not overlap but further including $X2$ would result in some shard losing more than one replica. Therefore, it responds with $approved = [X1, X3]$. Consequently, *TwineX* executes the updates for $X1$ and $X3$ and then notifies the TaskController. The TaskController subsequently responds with the next batch of tasks that are now safe to be updated, in this example, $approved = [X2]$.

Once all tasks in phase 1 of the `deploy` action are updated, Twine notifies Conveyor. Next, during the configured `bake` time, Conveyor periodically invokes the Health Check Service to access the health of $X1$, $X2$, and $X3$ (instead of all tasks in *JobX*), by comparing the error rate, latency, and user engagement metric of those tasks before and after the update.

If the health checks pass, Conveyor proceeds to phase 2 of the `deploy` action by instructing *TwineX* to update the remaining 3 tasks of *JobX* and instructing *TwineY* to update all 6 tasks of *JobY*. The process is similar to that in phase 1.

This example highlights a key principle in our design: separation of concerns. Conveyor orchestrates the execution of the deployment plan supplied by the service owner without worrying about how tasks are updated. A custom TaskController usually has a simple implementation since it only needs to determine which tasks can be safely updated together based on the application constraints. The complexity of actually updating tasks and managing their lifecycle is handled by Twine without the involvement of Conveyor or TaskController.

3 Deployment Scenarios and Solutions

To achieve universal coverage, Conveyor supports a wide range of deployment scenarios. This section presents these scenarios and the corresponding solutions in Conveyor.

3.1 Enabling In-place Updates

The software deployment approach affects hardware costs. The in-place update approach restarts an existing task on the same machine to run the new executable. In contrast, the mirroring approach, which is also called Red-Black [43] or Blue-Green [2] deployment, first starts a new job with the new executable, often on other machines, gradually redirects the traffic from the old job to the new job, and finally shuts down the old job. Although this approach is safer as the traffic can be quickly redirected back to the old job if the update fails, it needs extra hardware to run both the old and new jobs in parallel. Consider the example of deploying FrontFaaS to 500K machines every three hours. A naive mirroring approach would require 500K extra machines, which is unacceptable. One optimization is to divide it into many small jobs and utilize mirroring to update one small job at a time. However, this approach would result in the loss of quick rollback capability and complicate job autoscaling [16, 34]. Although further optimizations might be possible, the resulting solution would not necessarily be cheaper, simpler, or more generic than in-place updates.

Despite the benefits of hardware savings, the in-place update approach lacks widespread support from existing deployment tools due to the difficulty of ensuring deployment safety. Below, we present how we have made in-place updates safe and practical by co-designing our deployment tool (Conveyor) and our cluster manager (Twine [45]).

Collaborative control between Conveyor and Twine. As discussed in §2.2, during each phase of the `deploy` action, Conveyor instructs Twine to update a specific number or percentage of tasks, denoted as N_{big} and then waits for a period of time to collect comprehensive health signals before moving on to the next phase. Twine, however, does not update N_{big} tasks all at once. Instead, it updates only N_{small} tasks in one batch, where N_{small} is much smaller than N_{big}, to avoid losing too many tasks simultaneously. Twine then checks the liveness of each task before proceeding to update the next N_{small} tasks. Similar to other cluster managers [22], Twine's liveness check is rudimentary and only verifies that an individual task is running properly. However, it is unable to detect subtle issues such as the new code's memory regression compared to the old code, which is handled by Conveyor's comprehensive health checks (§3.2).

The collaborative control between Conveyor and Twine enables fast and safe deployments by assigning the most suitable functions to the right layers. Let's consider some alternative designs. If Conveyor instructs Twine to update N_{small} tasks instead of N_{big} tasks, and then waits for a period of time to collect comprehensive health signals, the deployment speed would be too slow. On the other hand, if Twine updates N_{big} tasks instead of N_{small} tasks in one batch, the service might lose too much capacity and become overloaded. Finally, eliminating Twine's rudimentary task liveness check would mean a destructive bug could affect N_{big} tasks instead of just N_{small} tasks before being detected.

Unlike the close collaboration between Conveyor and Twine, in the widely used open-source setup of Spinnaker [42] (deployment tool) instructing Kubernetes [22] (cluster manager) to update containers, Spinnaker cannot control the size of each phase, N_{big}, which defaults to the size of the entire job. This is because Kubernetes disallows partial-job updates. This simplistic approach is not suitable for in-place updates, because, when the comprehensive health checks detect a bug, it is likely that all tasks of the job have already been updated.

Pluggable TaskControl. In the task-update protocol described above, most services can use a per-service constant N_{small} and have no constraints on the specific tasks to be updated. However, the sharded key-value store described in §2.3 provides an example of services that require more precise control of N_{small} and the specific tasks to be updated. Figure 2 further illustrates how such a service can implement a custom TaskController to ensure safe in-place updates. TaskControl, in general, enables services to have precise control over task updates for various reasons, not limited to data shard availability. For example, FrontFaaS utilizes TaskControl to maximize its deployment speed (§4). Further details on TaskControl can be found in our previous work [24, 45].

Hardware failure and planned maintenance. When multiple tasks undergo in-place updates simultaneously, there is a risk of reducing the available capacity of a service to an unhealthy level. Merely controlling the update speed through the deployment pipeline is insufficient since certain tasks may be in an unhealthy state due to machine failures or planned maintenance, which Conveyor is unaware of. To tackle this issue, the service owner can inform Twine of a budget that indicates the maximum number or percentage of tasks that can be offline for any reason, such as task update, hardware failure, or planned maintenance. If the budget is projected to be exceeded, Twine pauses task updates.

Zero downtime hotswap. Our routing service in edge datacenters forwards user-facing traffic to our datacenters and holds live HTTPS connections to user devices. Naively restarting a task for an update would cause user-facing errors. Twine provides a same-host hotswap feature to solve this problem. It first starts a new task on the same machine, which binds to the same TCP ports as the old task. Then the new task and the old task cooperate with each other to hand over the live connections from the old task to the new task with the help of eBPF [14]. One limitation of hotswap is that it requires the container to be configured with sufficient memory to start two tasks, which is the reason why it is not used universally.

Summary. All of the aforementioned deployment scenarios with in-place updates cannot be properly implemented without support from the cluster manager. We believe this is a key reason why existing deployment tools primarily use the mirroring approach, since the cluster managers they rely on do not provide the necessary functions for in-place updates.

3.2 Deployment Safety

To enable continuous deployment, we accept bugs and deployment failures as a norm and rely on automated guardrails such as testing and health checks to detect release failures early and mitigate their adverse effects. In this section, we describe techniques that help Conveyor deploy safely.

Moving-target health checks. After each deployment phase, Conveyor invokes the Health Check Service (HCS) to evaluate the health of the service. Multiple health checks can be associated with a job, and each health check specifies a data source such as a time series database for monitoring data [31], a metric, data transformations (e.g., calculating specific percentiles), and a decision threshold. The metrics encompass system metrics (CPU, memory, crashes), RPC metrics (connections, errors), and application-level business metrics. The thresholds can be absolute (e.g., fail if CPU utilization exceeds 90%), A/B comparative (e.g., fail if the new task's CPU utilization exceeds that of the task that still has not been updated by 10%), or time-based (e.g., fail if the CPU utilization increases by 10% since the deployment started). By default, a failed health check triggers Conveyor to revert the release.

In contrast to traditional monitoring systems that track the health of an entire job, in-place updates require the HCS to track "*moving targets*", i.e., a dynamic subset of tasks that change throughout different deployment phases. Achieving this level of precision and adaptability necessitates a seamless integration between Conveyor, Twine, and HCS. Specifically, Twine assigns unique identifiers to tasks, enabling differentiation between the old and new tasks. The monitoring data for tasks is tagged with these identifiers. Depending on the current deployment phase, Conveyor dynamically instructs HCS to perform health checks on tasks with specific identifiers.

Code dependency analysis to prevent faulty releases. A monorepo [9] stores the code for an organization's many projects in a single repository, promoting code reuse but also leading to increased code dependencies. For instance, the ServiceRouter [37] library is compiled into nearly every service in Meta, and it may rely on a high-performance data structure library, which in turn may rely on a profiling library, and so on. In a monorepo setup, whenever a new version of a library is committed, any services that depend on the library will be automatically compiled with the new version. Consequently, the owner of a service may not even be aware that their service is affected by a bug in a shared library. To tackle this issue, Conveyor offers the Bad Package Detector (BPD). If library developers discover a bug in the library, they can report it to Conveyor. The BPD then utilizes a code dependency graph, which is provided by our build system [10], to identify and cancel the releases of all service executables that were built with the problematic version of the library.

Accurate code dependency analysis poses a challenge for the BPD as it requires finding the right balance between false negatives and false positives. Achieving perfect coverage would entail considering all possible direct and indirect dependencies of a service, which is often impractical. To strike a balance, the BPD currently tracks 14 levels of dependency. Our production data reveals that about 14% of to-be-deployed executables are invalidated by the BPD. This highlights the importance of handling bugs in dependent code.

Comprehensive testing. Conveyor supports various types of tests in deployment pipelines. The `PerfTest` tool records and replays production traffic to perform A/B testing between old and new code, while the `IntegrTest` [28] tool sets up interdependent services and tests their interactions. Additionally, `IntegrTest` can perform randomized fuzzing tests. `Canary` updates a small number of production tasks with new code and collects health signals, enabling direct testing in production. Multiple `canaries` can be executed in parallel to test different code variations. Even if a release's `deploy` action is not chosen for final execution (e.g., a release's `deploy` action, while waiting to start on Monday at 9 AM, gets superseded by a newer release), it is still valuable to execute the test actions to detect bugs as early as possible.

Summary. While other deployment tools also support testing and health checks, they lack some key capabilities. In contrast to HCS' ability to track "*moving targets*," traditional health checks rely on alarms defined for an entire job, which is insufficient to support in-place updates. Moreover, bug dependency analysis is not supported by existing deployment tools. Finally, we are not aware of any other large-scale adoption of record and replay as a generic platform for performance tests.

3.3 ML Model Deployment

Deployments of ML models for inference have emerged as an important issue, given the rapidly increasing number of ML applications. While existing deployment tools generally do not handle model deployments, Conveyor has been specifically enhanced to address this need and currently about 44% of its pipelines are for model deployments. Below, we describe Conveyor's support for model deployments.

Deployment via configuration change. In Conveyor's first implementation for model deployments, model update and executable update share the same pipeline, requiring Twine to restart the container. However, as model updates may occur more frequently than updates to inference executables, frequent container restart results in a frequent loss of expensive GPU capacity for request serving. Moreover, since a model often contains gigabytes (GBs) of data, the time required to load GBs of data during the restart can be lengthy.

To solve this problem, some inference services utilize two orthogonal pipelines. One pipeline deploys the inference executable through Twine, while the other deploys the model data through Configerator [44], Meta's configuration management system. To track model updates, all tasks serving a model subscribe to a configuration that specifies the current version of the model to be served. When a new version of the model becomes available, instead of exposing it to all tasks simultaneously, Conveyor instructs Configerator to incrementally expose the new version to tasks in phases, following the deployment pipeline. Configerator utilizes a data-distribution tree to notify the tasks in a scalable manner. Once the tasks receive the notification of a new model version, they utilize Owl [15], a peer-to-peer data-distribution system, to fetch the new model. While still serving live requests, a task merges the new model into the old model piece by piece without consuming additional memory as it never keeps full copies of both models in memory simultaneously. Overall, Conveyor ensures safe deployments of models through phased releases, which are meticulously managed via configuration changes.

Conveyor's ability to perform phased deployments of generic configuration changes extends beyond its use in ML model updates. Conveyor pipelines are widely utilized to ensure safe deployments of various configuration changes.

Lockstep deployment of interdependent services. Some of our ML models are too large to fit in one machine's memory, so they are partitioned into interdependent shards, each including multiple replicas for fault tolerance and throughput. Each shard is mapped to a different job, and typically the first shard serves as the aggregator to combine results from other shards. However, updating different shards independently may cause compatibility issues, because combining outputs from different versions of the shards will produce incorrect results. To ensure compatibility, replicas of the first shard are configured to only receive outputs from replicas of other shards of the same version. This design requires Conveyor to perform a lockstep deployment of different shards to avoid capacity loss during deployment. For instance, 5% of each shard's replicas are updated at the same time and 5% of the client traffic is directed to the new version, before proceeding to update 10% of each shard's replicas, and so forth.

Parent-child pipelines. Meta's ML inference system serves tens of thousands of ML models using about 10 different inference executables. Each model, along with its inference executable, is deployed via a separate pipeline. Since many models share the same inference executable, updating one executable may cause thousands of pipelines to initiate a new release at the same time. This not only causes a load spike on Conveyor and Twine, but also increases the risk of an undetected bug in the inference executable impacting many models simultaneously.

While existing deployment tools manage each pipeline in isolation, Conveyor coordinates releases across pipelines that share common artifacts by setting up a parent-child relationship between them. Specifically, each inference executable is managed by a parent pipeline, which includes sophisticated testing but no `deploy` action, while the pipelines for models served by the executable act as its child pipelines and include the `deploy` action. When updating an ML model without modifying the executable, only the corresponding child pipeline is executed, without involving the parent pipeline. However, when updating the executable, the parent pipeline is executed first. If successful, all the corresponding child pipelines are then executed with randomized delays to avoid starting them at the same time and overloading the system. Moreover, it can be configured in such a way that a subset of child pipelines for less important models is executed first to help detect issues with the executable.

3.4 Advanced Features for Universal Adoption

To achieve universal adoption, Conveyor must support advanced use cases. We describe them in this section.

DAG pipelines. Conveyor initially modeled deployment pipelines as a sequence of sequential stages, such as build→test→deploy. However, this pattern has not generalized to complex services like FrontFaaS, which may build, test, and deploy two different versions in parallel (§4). The sequential pipeline is overly restrictive in that it requires testing for both versions to finish before the deployment for any version can start. Therefore, we have improved Conveyor by modeling its pipelines as directed acyclic graphs (DAGs). These DAGs support conditions, branching, and mutual exclusion groups. Since multiple releases of a pipeline may be executed in parallel, Conveyor allows users to define actions as a mutual exclusion group, meaning that concurrent releases should not execute these actions in parallel. For example, if a pipeline includes a `load-test` action and a `deploy` action, they may be put in an exclusion group so that a `deploy` action in one release will not accidentally fail its health check due to a `load-test` from another concurrent release.

Mutable artifacts. Conveyor initially mandated one `build` action at the beginning of a pipeline, resulting in an immutable artifact, the executable, to be used throughout the pipeline. However, this simple model does not fit well with complex deployment scenarios. One example is feedback-directed optimization (FDO) [12], which involves profiling executables in production and using the profiling data to guide recompilation of the code, resulting in an updated artifact, the new executable. Therefore, Conveyor has been extended to support mutable artifacts and allow multiple `builds` within one pipeline. To perform FDO, for example, the pipeline can first build the baseline executable, deploy it to a small number of tasks that receive production traffic, and collect profiles. It then builds the executable again using FDO and finally deploys the optimized executable to all tasks.

CLI and daemon deployment. Every machine in our fleet runs a set of CLI tools and daemons that provide utility functions. As these host-level executables are not managed by Twine, Conveyor provides a `deploy` action type called *Slowroll* to manage them. Due to the massive scale of deploying these executables to every machine, Slowroll adopts a pull model instead of Twine's push model. In the pull model, each machine periodically downloads the new version of the software being deployed. These deployments can take a long time to finish. For example, due to the massive scale of deploying a specific daemon to every machine, its pipeline has 43 phases and a deployment can take more than a week. Moreover, some CLI tools are infrequently used, requiring a significantly longer "bake time" (e.g., 8 hours) to collect health signals.

3.5 Software Backward Compatibility

Despite backward compatibility being a generic requirement for software deployment, it is largely left for services to handle because it often involves application-specific logic.

API backward compatibility. During a deployment, the new and old versions of a service will coexist for a while. When the API of a service needs to be changed, a common practice at Meta is to support both the old and new APIs and gradually switch clients to invoke the new API. Once all clients are migrated to the new API, the code for the old API can be deleted. This is called *N*-1 compatibility, meaning that in addition to the current version *N* of the API, it also supports version *N*-1 but not version *N*-2 or older.

Database backward compatibility. When a service update involves data migration from one database to another or an upgrade of the database schema, extra care is needed. For example, when switching from an old database to a new database, a common strategy is to perform double writes: the service writes new data to both databases but always reads data from the old database, while a background process migrates old data from the old database to the new database. This strategy simplifies the rollback process as it only requires deleting the new database and redeploying the old code.

3.6 Summary of Distinguishing Features

While Conveyor and Twine provide many features to achieve universal adoption, we consider that the following features distinguish Conveyor and Twine from existing tools most.

1. In-place updates allow us to minimize hardware costs, which is important while operating at hyperscale.
2. TaskControl provides flexibility for services to precisely control the speed and sequence of their task updates.
3. The ability to update a subset of tasks and perform health checks on moving targets enables Conveyor to exert fine-grained control over tasks, enabling in-place updates.
4. In a monorepo [9] setup, it is important to perform code dependency analysis to prevent faulty releases.
5. Features to support ML model deployments have grown to become a first-class citizen in Conveyor.
6. Conveyor provides a one-tool-fits-all solution for safe deployments of various artifacts, such as service executables, ML models, and application configurations.

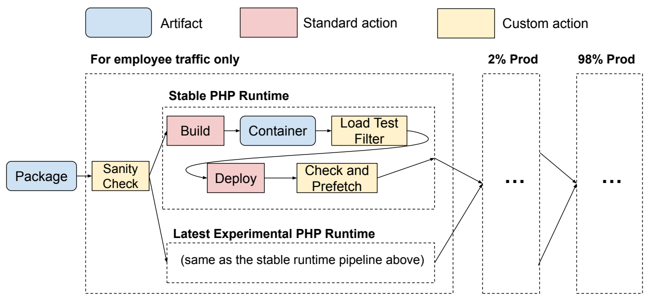


Figure 3: Simplified deployment pipeline for FrontFaaS.

Existing deployment tools lack these features. Without them, Conveyor would not have achieved universal adoption.

Conveyor is designed to be a generic and extensible deployment tool, and does not burden itself with every feature needed by every service. Its extensible architecture allows services to easily implement their own TaskControllers, actions, and new types of artifacts. Moreover, sometimes it is preferable to consider a service redesign to leverage Conveyor's standard functions. For instance, Meta's ML inference platform initially hosted multiple ML models as separate processes within a single container, and requested Conveyor to provide the feature of using independent pipelines to update different processes within the same container. Since this feature is complex and not needed by other services, we did not support it in Conveyor and the ML inference platform was ultimately redesigned to adopt the one-model-per-container approach.

4 Case Study of FrontFaaS

To illustrate the end-to-end usage of Conveyor, we present a case study of FrontFaaS, a serverless Function-as-a-Service (FaaS) platform for PHP functions. It differs from other serverless platforms such as AWS Lambda [5] in several ways: 1) it only hosts synchronous functions that clients directly invoke, while event-driven asynchronous functions are hosted by another platform; and 2) for efficiency, a single PHP runtime process can execute multiple functions concurrently. Since FrontFaaS is serverless, developers simply commit function code without worrying about code deployment or server provisioning. Each year, tens of thousands of developers commit serverless function code to FrontFaaS, with thousands of code commits each workday. Currently, FrontFaaS runs on over half a million machines and makes a new release every three hours to deploy the code of all functions together [33]. Through FrontFaaS, Meta developers create PHP functions that are servicing traffic from end users when they visit Meta web pages. Note that many Meta employees are end users of Meta products as well, and their traffic is often used for testing purposes as discussed next.

Figure 3 shows a simplified deployment pipeline for FrontFaaS. To enable phased deployments, the machines hosting FrontFaaS are partitioned into three pools: one for servicing employee traffic, one for servicing 2% of production traffic, and one for servicing the remaining production traffic. Accordingly, the deployment pipeline is divided into three major phases, one for each machine pool. A release proceeds to the next machine pool only if the deployment to the prior pool succeeds. Each pool is split into two sub-pools: one processing a small amount of traffic with an experimental version of the PHP runtime [30] and one processing the remaining traffic with a stable version. If the experimental version outperforms the stable version, it will become the new stable version.

The first step in the deployment pipeline is a custom `sanity-check` action that performs a FrontFaaS specific logic to ensure that there are no deployment-blocking alerts. Then, a standard `build` action is used to build the container image. Independent of specific releases, load tests always continuously run on certain machines to gather performance data and build capacity models. The custom `load-test-filter` action excludes machines that are scheduled to be load-tested from being monitored by health checks, to avoid load-test induced false alarms during health checks. The pipeline then uses a standard `deploy` action to update tasks and run health checks. Finally, a custom `check-and-prefetch` action runs the sanity check again while asking Twine [45] to prefetch FrontFaaS' container image on machines that will be updated soon. This prefetch reduces the overall duration of each release by 5-30%.

Since FrontFaaS continuously deploys every three hours across more than half a million machines, fast deployment is an important requirement. Within a `deploy` action, FrontFaaS relies on two techniques to accelerate a deployment. First, it implements a custom TaskController that controls the rate at which tasks are updated based on the CPU utilization of FrontFaaS jobs. It tries to concurrently update as many tasks as possible, as long as the temporary loss of those tasks does not cause other FrontFaaS tasks to handle too much traffic and become overloaded. During off-peak hours, updates can be applied to many tasks in large batches, while during peak hours when traffic is high, the Task Controller applies updates in smaller batches to prevent overload. Thanks to this optimization, deployments during off-peak hours are approximately three times faster than those during peak hours.

The second technique to accelerate a deployment is to update tasks and perform health checks in parallel, but the short health-check time requires FrontFaaS to have highly accurate health checks. FrontFaaS primarily relies on three health-check metrics: 1) the number of fatal errors, 2) the number of unavailable tasks, and 3) the write rate of error logs. Every minute during a `deploy` action, these metrics are averaged over the previous three minutes and compared to the average during a three-minute period before the deployment began. If any datacenter region experiences an increase in these metrics above a certain threshold, Conveyor pauses all updates in that region. If too many jobs are paused, Conveyor considers the deployment a failure and initiates a rollback.

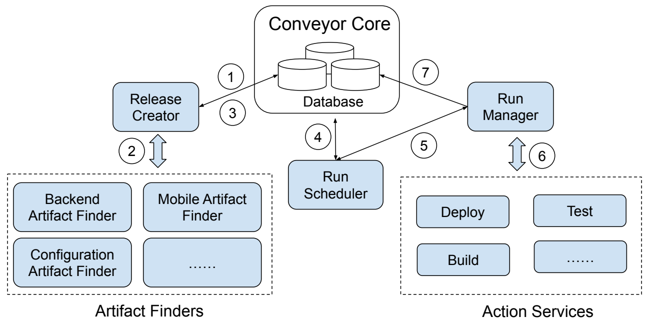


Figure 4: Conveyor Architecture.

Overall, FrontFaaS achieves fast and safe deployment at hyperscale by utilizing TaskControl and multiple custom actions made possible by Conveyor's extensible architecture.

5 Design and Implementation of Conveyor

In this section, we briefly summarize Conveyor's design and implementation.

5.1 Conveyor Design

Figure 4 depicts the architecture of Conveyor. Conveyor comprises several stateless services that share a database. The database stores user-defined pipelines, metadata, as well as the last step executed for each release and action. To accommodate the high throughput due to tens of thousands of pipelines, the database is partitioned into many shards.

For robustness and scalability, all components of Conveyor adhere to the worker-pool paradigm. They store the IDs of pipelines, releases, and action runs in a queue. Multiple workers then periodically scan the queue, identify the required operations, and execute them. The number of workers can be scaled up or down based on the load. The execution of a pipeline involves the following steps.

Create release (Steps 1-3 in Figure 4). Each pipeline accepts a set of input artifacts. The Release Creator periodically scans for new input artifacts (Step 1) by invoking Artifact Finders (Step 2). When new artifacts are discovered, Conveyor creates a "release object" in the database, thereby triggering the execution of the pipeline (Step 3). We provide seven standard Artifact Finders, and allow users to implement custom Artifact Finders. All Artifact Finders implement a single method, `getArtifacts()`, which returns a set of discovered artifacts. For example, one Artifact Finder identifies latest commits that successfully passed unit tests, while another identifies executables that have been already built and marked with a specific tag.

Schedule (Step 4 in Figure 4). The Run Scheduler schedules the execution of releases and actions by periodically scanning through all releases and stepping through each action in each release's corresponding pipeline. When the input artifacts for an action exist and all preceding actions have successfully completed, the Run Scheduler schedules the action for execution.

Because Conveyor allows for multiple active releases derived from one deployment pipeline, the Run Scheduler must coordinate actions across these releases. Some actions, like `canary`, permit concurrent execution on multiple releases, while others, like the `deploy` action, only allow execution on a single release at a time. Therefore, the Run Scheduler must determine which active release executes the `deploy` action. Consider a case where a pipeline deploys to production on every Monday at 10 AM. If multiple code changes have occurred since the last deployment, two releases might be ready to run the `deploy` action on Monday at 10 AM. In this case, the Run Scheduler will cancel the older release.

Run actions (Steps 5-7 in Figure 4). The Run Scheduler determines when an action should be executed and notifies the Run Manager accordingly (Step 5). The Run Manager then initiates the execution of the action by invoking a service that implements the action's logic (Step 6). When executing an action, the Run Manager first invokes the action's `startRun()` method and then periodically calls its `getRunProgress()` method to track the progress. The outcome of the action is recorded back to the database (Step 7).

Conveyor offers several standard actions, including the `build` action for compiling source code and running corresponding unit tests, various testing actions, the `pkg` action for tagging executables for easy naming and access, the `deploy` action for deploying the new version to production, the `CustomScript` action for running any user-defined script (e.g., to shift traffic before a `deploy` action starts), and the `ManualPick` action for pausing a pipeline and awaiting the service owner's decision. Conveyor's extensible architecture allows users to create custom actions by implementing the `startRun()` and `getRunProgress()` methods.

5.2 Implementation of the Deploy Action

Since the `deploy` action is the most complex part of Conveyor, we describe it in more detail below. Like a sub-pipeline within the greater Conveyor pipeline, the `deploy` action's deployment plan specifies the number of phases, which jobs and tasks to update in each phase, the baking time after each phase, and the success criteria. Conveyor supports both percentage and task-count based configuration when specifying the success criteria and the tasks to be updated.

A few widely used deployment plans are commonly employed. For small services or services that prioritize deployment speed, a common plan is to request Twine to update all of their tasks in a single phase. Therefore, the number of tasks to update in a phase, N_{big}, as described in §3.1, equals the total number of tasks in the job. However, please recall from §3.1 that Twine is still configured to update only N_{small} tasks at a time to avoid losing too many tasks simultaneously. For services that prioritize safety, their jobs are often updated region by region, as our services are always designed to tolerate the loss of a whole region. To balance safety and speed, a common strategy is to update an exponentially increasing number of tasks in each phase (e.g., 1% of tasks in the first phase, 10% in the second, and 100% in the last), assuming that a bug is likely to be revealed in the early phases.

For services requiring high deployment speed, Conveyor can update tasks and perform health checks in parallel. As described in §3.1, during a deployment phase, Conveyor by default requests Twine to update a group of N_{big} tasks, waits for the updates to finish, and then periodically performs health checks during the bake time. The whole process can take a long period of time. With the parallel approach, Conveyor submits the request for Twine to update N_{big} tasks and then immediately performs health checks while the update is still in progress. This approach saves the wait time but necessitates the service to have highly accurate health checks due to the short health-check time.

Finally, users have the option to deploy to environments that are not managed by Twine by implementing custom *deployment types*. For instance, we maintain a pull-based deployment type for CLI tools and Linux daemons running on bare-metal machines (§3.4). Furthermore, users have created over 20 custom deployment types for various non-standard deployment targets, such as VMs in public clouds, application configurations, and serverless stream-processing functions.

To implement a custom deployment type, a user needs to build a service that implements a few methods. When initiating a new `deploy` action, the `fetchDeployUnits()` method is invoked, returning a list of *deploy units*. Each deploy unit represents a group of tasks to be updated. Then the `executeUpdates()` method is called to update a set of deploy units with a specific set of artifacts for deployment. Finally, the `deploy` action periodically calls the `trackUpdates()` and `runHealthChecks()` methods until all updates are completed. If any health checks fail, the `deploy` action fails early.

5.3 Availability, Reliability, and Recoverability

Currently, Conveyor is implemented in 360K lines of Rust code, including test code and utility tools, and its components run on several hundred machines. Conveyor is not the performance bottleneck in software deployment, as other tools that Conveyor relies on, such as Twine, often perform more extensive work than Conveyor itself.

For high availability, both the database used by Conveyor and each Conveyor component are replicated across multiple datacenter regions. However, it is not necessary to replicate every Conveyor component in every region. Conceptually, one global setup of Conveyor manages the deployments of all services across all regions. A Conveyor component in region X can communicate with the Twine instance in region Y to remotely drive the deployment of services in region Y.

Presently, Conveyor's service level objective (SLO) is to ensure that less than 0.5% of deployments fail due to issues in Conveyor or any of its dependencies, such as Twine, Health Check Service, build service, and Configerator. Conveyor consistently meets or exceeds this SLO.

The circular dependency between Conveyor and Twine poses challenges to their recoverability. Conveyor consists of a set of services that are deployed via Conveyor itself and hosted inside containers managed by Twine. Similarly, Twine is also implemented as a set of services that are deployed via Conveyor and hosted inside containers managed by Twine itself. When Conveyor or Twine fails, the entire ecosystem cannot update itself. To address this issue, we have designed them to be self-recoverable whenever possible and have introduced manual recovery tools for worst-case scenarios. In the event that Conveyor fails and cannot deploy bug fixes for itself, direct commands can be issued to Twine to start new jobs for Conveyor with the proper bug fixes.

To set up Twine to manage itself, we have implemented a two-layer deployment approach. The top layer consists of two independent instances of Twine, and under normal conditions, one instance can manage and update the other. These top-layer instances are responsible for managing and updating the numerous Twine instances in the bottom layer, which in turn manage and update user jobs. As long as at least one of the top-layer's Twine instances is functioning properly, all Twine instances can be updated normally. In the event that both top-layer Twine instances experience malfunctions, we have a dedicated tool that can be used to directly bootstrap a top-layer Twine instance and initiate the recovery process.

5.4 Lessons from Conveyor's Evolution

Over the course of nine years, Conveyor has undergone significant evolution, progressing from v1 to v2, and eventually to v3. Each subsequent version represents a complete system rewrite that incorporates the valuable lessons we have learned.

As a CLI tool, Conveyor v1 enables service owners to manually initiate phased in-place updates of services. Service owners could define the rollout phases and health checks. The Health Check Service (HCS) was developed along with Conveyor from the very beginning. Despite Conveyor v1 being relatively simple, about 12% of services adopted it, demonstrating a strong need for a standard deployment tool.

The biggest change from Conveyor v1 to v2 was to make it a long-running service so that it could automatically start deployments on a pre-configured schedule without manual intervention. We also introduced a more complete pipeline model, where a pipeline consisted of a sequence of phases, each comprising a set of actions. Several enhancements were implemented, including support for the Bad Package Detector (BPD), parent-child pipelines, and pull-based deployments. To boost adoption, Conveyor and Twine were made extensible by providing various interfaces for custom integrations, such as TaskControl, custom actions, and custom artifact finders. As a result of these features and the company-wide Push4Push program, about 94% of services adopted Conveyor.

The remaining 6% of services that did not adopt Conveyor were the largest and most complex ones, requiring more advanced features. Furthermore, the rapid adoption of Conveyor exposed its limitations in terms of performance and reliability. As a result, Conveyor v3 was introduced as another complete rewrite, this time switching from Python to Rust. The data model evolved from a sequential pipeline to a DAG. Additional features were implemented, including lockstep deployment, mutable artifacts to support feedback-directed optimization (FDO), deployment of mobile apps and application configurations, and better support for ML models and FrontFaaS. These enhancements helped Conveyor achieve universal adoption.

One ongoing evolution of Conveyor v3 is to enhance its real-time responsiveness. In Conveyor's current architecture, the latest state of each release is stored in a database, and a group of Conveyor workers periodically poll the database to identify actions that are ready for execution. We chose this periodic polling design for its robustness. However, the deployment of ML models and application configurations now requires faster execution of pipelines that cannot be supported by the polling method. Therefore, in Conveyor's new design, the completion of one action will immediately trigger the execution of the next action without any delay. However, we will still maintain the polling mechanism as a reliable fallback to safeguard against any transient failures.

6 Evaluation in Production

In this section, we use production data to help answer the following questions:

1. Has Conveyor achieved universal coverage?
2. Do developers trust fully automated deployments?
3. Are Conveyor's deployment-safety mechanisms effective?
4. How often do deployments fail and why do they fail?
5. What are the observed patterns in pipeline setup, and what are the best-practice recommendations for pipeline setup?

Using three weeks of data from April to May 2023, we studied all deployment pipelines, which amounted to more than 30,000 pipelines. We divide them into four categories:

- `Regular services`: 24.4% of the pipelines deploy traditional non-ML services through containers managed by Twine [45]. This category serves as the primary point of comparison with other deployment tools, as those tools may not support the other categories listed below.
- `Large services`: 0.45% of the pipelines deploy the largest services that consume 80% of our fleet's total capacity, with each of them using at least 7,700 servers. These `large services` are a subset of `regular services`.
- `ML models`: 44.4% of the pipelines deploy ML models, predominantly through Twine, with some utilizing Configerator [44] to control when a task consumes a new model.
- `Other pipelines`: The remaining 31.1% of pipelines are used for various purposes, such as running tests without performing an actual deployment. The data we report for this category will only include those with at least one `deploy` action, which amounts to 6.7% of all pipelines. These pipelines deploy artifacts that are not managed by Twine, such as CLIs, daemons, configurations, and mobile apps.

6.1 Universal Coverage

It is challenging to precisely calculate the percentage of services that should utilize Conveyor but do not use it, primarily due to the presence of experimental services that will never be deployed in production. Interviewing the owners of tens of thousands of services to determine this percentage would be impractical. Instead, we focus on calculating the coverage for all 195 `largest services`. These services tend to be complex, making the adoption of Conveyor more challenging compared to other services.

Conveyor achieves 100% coverage for these large services, with the following caveats: 1) One of them is an ML training job, where software updates during its training run are intentionally avoided. 2) Five of them are short-lived experimental services that do not need automated deployments as they will not be deployed into production. Overall, the advanced features described in §3 enable Conveyor to achieve universal coverage, even for the most complex services.

6.2 Trust in Fully Automated Deployments

To understand whether developers trust fully automated deployments, we classify pipelines into five categories based on their release schedules: 1) *Continuous deployment*, which is executed immediately whenever the input artifacts are ready; 2) *Daily*, which is executed at least once every working day at a fixed time, such as 9AM; 3) *Weekly*, which is executed at least once each week; 4) *Biweekly/monthly*, which is executed biweekly or monthly; 5) *ManualPick*, which may automatically execute early actions such as small-scale deployments, but requires human confirmation before executing the final stage of large-scale `deploy` actions.

Table 1 shows that among `regular services`, 54.6% adopt continuous deployments, and in total 96.5% adopt fully automated deployments without manual validation. These results indicate that with the guardrails provided by testing, health checks, and automated revert of faulty releases, developers release often and trust fully automated deployments.

Although only 71.8% of `large services` adopt fully automated deployments, it already demonstrates a high degree of trust in deployment automation, considering that they are complex and hyperscale services serving billions of users.

	Continuous	Daily	Weekly	Biweekly/Monthly	ManualPick
Regular	54.6%	24.8%	16.8%	0.4%	3.5%
Large	16.0%	16.8%	37.4%	1.5%	28.2%
ML	99.9%	0.0%	0.0%	0.0%	0.0%
Other	48.5%	26.3%	19.0%	0.3%	5.9%

Table 1: Classification of pipelines based on their schedules.

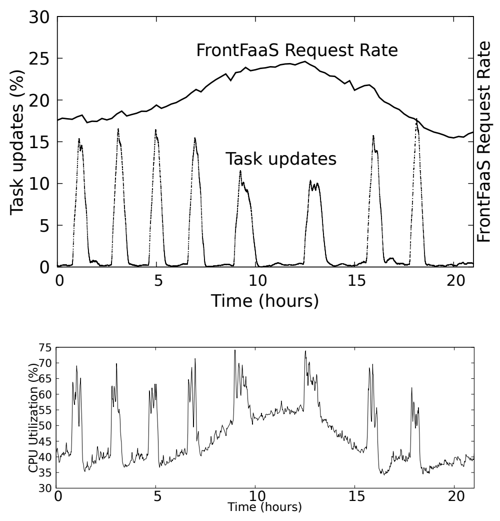


Figure 5: TaskControl slows down the deployment speed for FrontFaaS during peak hours.

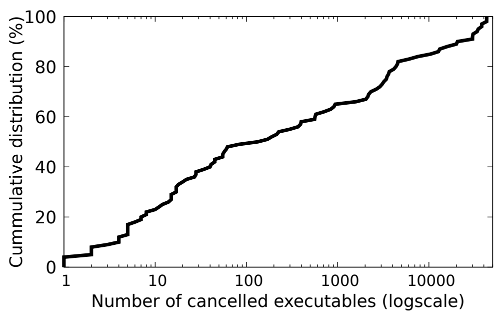


Figure 6: CDF of the number of executables canceled by the Bad Packet Detector (BPD) per reported bug.

	Regular services	Large services	ML models	Other pipelines
Successful	27.5%	15.4%	33.7%	38.0%
Failed	13.8%	20.9%	5.2%	4.2%
Canceled	58.7%	63.7%	61.1%	57.8%

Table 2: Breakdown of the end states of releases.

One main reason for the remaining ones using manual validations is that they have highly complex health checks, making it difficult to achieve both a low false positive rate and a low false negative rate simultaneously, regardless of how the decision threshold for health checks is set.

Although the deployments of some services are already fully automated, they still prefer deployments on a fixed schedule (daily or weekly) over continuous deployment for several reasons. First, a failed deployment of a complex service may cause a partial outage in production, which can take hours to mitigate. Therefore, it is preferred to start the deployment at a fixed time in the morning to ensure that incident mitigation does not extend into the night. Second, although almost all services are fault-tolerant against task updates, they may experience degraded SLOs during a task update. For example, updating a ZooKeeper ensemble will trigger a leader re-election and result in delays in responding to client requests. These services prefer to avoid degraded SLOs caused by frequent deployments of every single code change.

6.3 Deployment Safety at Hyperscale

To ensure the safety of in-place updates, Twine's TaskControl API allows services with special needs to exert precise control over their task updates. For example, FrontFaaS' custom TaskController dynamically adjusts the deployment speed in order to safely and continuously deploy every three hours across more than half a million machines. Figure 5 illustrates what happens in a region that runs over 10,000 FrontFaaS tasks. The top figure shows the normalized request rate for FrontFaaS, along with the percentage of updates to FrontFaaS tasks in that region. The bottom figure shows the average CPU utilization of those FrontFaaS tasks. During the site's peak hours, which occur between hours 8 and 15, FrontFaaS' TaskController instructs Twine to reduce the number of task updates in order to prevent the temporary loss of too many tasks and avoid overloading the system. Consequently, it takes longer to complete a release during these peak hours. Despite the fluctuating load on the site and the load spikes caused by task updates (see the bottom figure), the overall CPU utilization remains below 75%. Without TaskControl, this level of application-specific adaptation and precise control is hard to achieve with other deployment tools.

To ensure the safe deployment of services that share a monorepo [9], which often entails complex code dependencies, developers can report bugs to Conveyor. The Bad Package Detector (BPD) automatically identifies the affected executables scheduled for deployment and cancels their releases. While only approximately one such bug is reported to Conveyor per day, their impact tends to be widespread. Figure 6 illustrates the number of executables affected by these bugs. Around 15% of these bugs impact over 10,000 executables. Certain bugs, such as those found in Meta's RPC library [37], have the potential to impact every service. Due to the broad impact of these bugs, the BPD cancels approximately 14% of all executables scheduled for deployment. This extensive impact highlights the need for automated code dependency tracking in a monorepo to ensure deployment safety.

6.4 Deployment Failures

To understand how often deployments fail and why they fail, we analyze the failure data of releases and `deploy` actions.

6.4.1 Release Failures

Over the three weeks of our evaluation, Conveyor generated millions of releases. Table 2 summarizes the end states of these releases. Release cancellations commonly occur when a new release supersedes an old release that was not yet deployed. While most releases are canceled, they are still highly valuable as they facilitate the execution of builds and tests, aiding in the early detection of bugs.

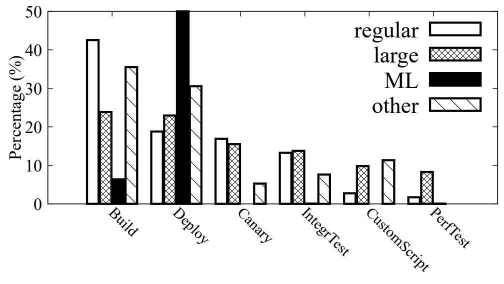


Figure 7: Breakdown of release failures by action types.

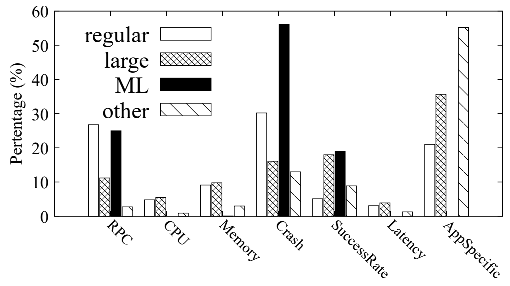


Figure 8: Breakdown of health check failures.

Although the release failure rate seems high, it actually indicates the effectiveness of deployment automation. Let's consider a simple pipeline consisting of `build`→`test`→`deploy`. If a release fails during the `build` or `test` action, it means that the problem is detected early, preventing a faulty release from reaching production. This is precisely why, despite the high failure rate, developers maintain a high level of trust in deployment automation (§6.2).

We show the breakdown of release failures by action types in Figure 7. As an example, the value of "*40%*" for the "*regular*" bar in the `build` category does not mean that 40% of `builds` fail for `regular services`. Instead, it means that out of all failed releases for `regular services`, 40% of them failed while executing the `build` action. The value of the "*ML*" bar in the `deploy` category is 93.6%, but we cap it at 50% to make other bars more visible.

Figure 7 shows that, except `ML models`, the majority of release failures are detected by standard builds and tests (`canary`, `IntegrTest`, and `PerfTest`). Note that `builds` involve running unit tests and will fail if any tests do not pass. For `ML models`, the setup of parent-child pipelines between inference executables and models (§3.3) helps reduce failed releases. During the three weeks, nine failures occurred in the parent pipelines. Among them, eight were detected by unit tests, and one was detected by `PerfTest`. These failed parent pipelines did not trigger the execution of the corresponding child pipelines. Otherwise, the number of failed child pipeline releases would have increased by about 50%.

6.4.2 Failures in `Deploy` Actions

Failures in `deploy` actions occur at the important stage of updating tasks and could lead to user-visible errors. The failure rates of `deploy` actions vary across pipeline types: `regular services` (5.4%), `large services` (6.2%), `ML models` (14.3%), and `other pipelines` (1.1%). In all cases, health check failure is the top reason for `deploy` failures, followed by update timeout, which is typically caused by too many unhealthy tasks during a deployment.

Figure 8 presents the breakdown of health check failures. While system-level metrics, such as CPU, memory, crashes, and RPC errors, can help identify many problems, "*AppSpecific*" metrics still play a significant role. Frequently used such metrics include the increase in the number of specific errors returned to users, increase in the number of retries, decrease in correctness metrics, or changes in user engagement.

	Regular services	Large services	ML models	Other pipelines
False negative	0.92%	3.14%	0.012%	0.12%
False positive	11.5%	41.3%	0.0025%	17.4%

Table 3: False positives and false negatives of health checks.

Developers often need to make a tradeoff between false positives and false negatives when using health checks to detect bugs because health anomalies can also be caused by other factors such as hardware failures or changes in workload. To approximate the rate of false negatives (i.e., bugs not being detected), we calculate the percentage of releases that finished successfully but were later reverted or patched by developers. However, since developers sometimes patch or revert a release for purposes other than fixing bugs (e.g., to do a quick test), these numbers should be viewed as an upper bound for false negatives. To approximate the rate of false positives (i.e., health anomalies in the absence of a bug), we calculate the percentage of `deploy` actions that reported health anomalies but were allowed to proceed by a human.

As shown in Table 3, except for `ML models`, the occurrence of false positives is significantly higher than that of false negatives. Notably, the rate of false positives for `large services` reaches as high as 41.3%. This is because health checks for large services are typically more intricate, and developers tend to use stringent health check thresholds to ensure release safety. When faced with health anomalies, they prefer to rely on manual investigations to determine whether to proceed with a release or not.

Figure 9 further presents the point at which a `deploy` action fails. The *progress* metric is calculated as the number of tasks that are supposed to be updated till the end of the failed phase divided by the total number of tasks to be updated in all phases. We exclude single-phase `deploy` actions from this figure, since their progress is either 0 or 1. Figure 9 reveals a bimodal pattern, where failures occur either very early or very late. While many bugs can be detected when only a small subset of tasks is updated, some bugs, such as subtle performance regression, can only be detected after they are deployed at scale. These kinds of bugs pose the most challenging problem for software deployment.

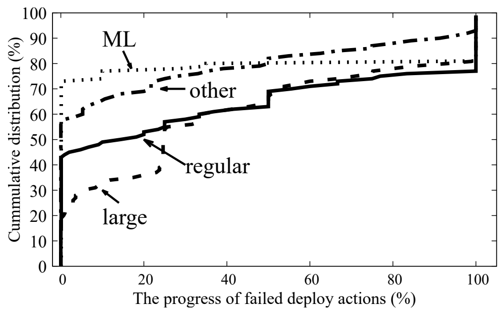


Figure 9: The progress of failed `deploy` actions.

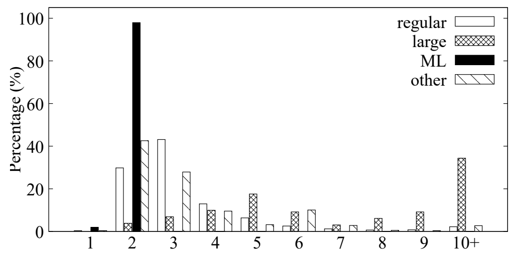


Figure 10: Number of actions per pipeline.

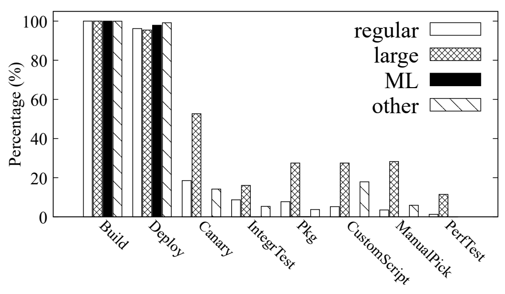


Figure 11: Percentage of pipelines that use a specific action.

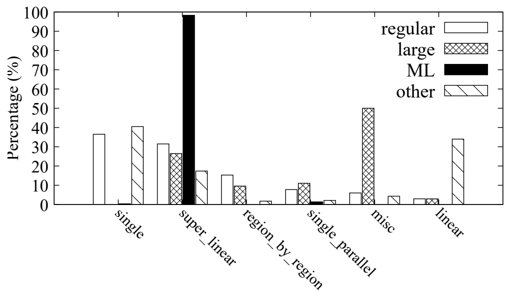


Figure 12: Breakdown of different `deploy` patterns.

6.5 Pipeline Patterns & Recommendations

In this section, we analyze various aspects of pipeline statistics to gain a better understanding of how pipelines are used in production. Based on our findings, we provide best practices for pipeline design in §6.5.4.

6.5.1 Pipeline Configuration

Figure 10 shows that the average number of actions per pipeline varies: `regular services` (3.5), `large services` (12.2), `ML models` (2.0), and `other pipelines` (4.0). As expected, `large services` have much deeper pipelines. For `ML models`, all child pipelines (§3.3) use a uniform setup with two actions: `build` and `deploy`. The parent pipelines for inference executables have more actions such as `PerfTest`, but since the number of child pipelines is about 1,000 times larger than that of parent pipelines, the statistics here mainly represent those of child pipelines.

We further examine the popularity of different types of actions, represented as the percentage of pipelines that include at least one corresponding action type. As shown in Figure 11, `build` and `deploy` are the two most popular actions, as expected. We observe two distinguishing characteristics of `large services`. First, they are more likely to include tests (`canary`, `IntegrTest`, and `PerfTest`). Second, they rely more on human decisions (i.e., `ManualPick`) to determine whether to proceed.

6.5.2 Deploy Action Configuration & Runtime Statistics

To balance safety and speed, a `deploy` action often consists of multiple phases, each updating a subset of tasks. We observe a few popular patterns in the setup of `deploy` actions: 1) the "*super linear*" pattern, which updates a small percentage of tasks in the first phase and updates a higher percentage in later phases; 2) the "*single*" pattern, which updates all tasks in one phase; 3) the "*single parallel*" pattern, which uses one phase but updates multiple jobs in parallel; 4) the "*region-by-region*" pattern, which updates one region's jobs per phase; and 5) the "*linear*" pattern, which updates the same percentage of tasks in each phase. By definition, these patterns are not exclusive. For example, a *region-by-region* pattern could be *linear* as well. To separate them, we use the following rule: *single parallel* > *single* > *region-by-region* > *linear* > *super linear*. It means that, if a *deploy* action meets more than one definition, we categorize it as the first one in the chain.

As shown in Figure 12, the setups are very diverse. Among `regular services`, simple ones prefer the *single* pattern and complex ones prefer the *super linear* pattern to balance safety and speed. `Large services` employ various *misc* patterns, with a concrete example shown in §6.5.3. `ML models` mostly use the *super linear* pattern. Among `other pipelines`, simple ones prefer the *single* pattern and complex ones prefer the *region-by-region* pattern for safety.

The execution time of `deploy` actions has a long-tail effect: for `regular services`, its P50 is 2.3K seconds and P99 is 86K seconds; for `large services`, its P50 is 2.0K seconds and P99 is 186K seconds (52 hours); for `ML models`, its P50 is 2.9K seconds and P99 is 9.8K seconds; for `other pipelines`, its P50 is 0.33K seconds and P99 is 15K seconds. We will elaborate on the long `deploy` time in §6.5.3.

Table 4 further decomposes `deploy` action's execution time into four components: *task update*, *bake*, *preprocessing*, and *postprocessing*. Recall that the bake time is the time after all updates in a phase have been completed, during which Con-

	Regular services	Large services	ML models	Other pipelines
Task update	21.2%	45.1%	39.2%	31.4%
Bake	31.7%	25.1%	58.1%	29.6%
Preprocessing	18.7%	2.9%	1.9%	5.4%
Postprocessing	28.4%	27.0%	0.9%	33.6%

Table 4: Time spent in different stages of `deploy` actions.

veyor periodically checks the health of the updated portion of a service. *Preprocessing* and *postprocessing* refer to custom operations performed before and after updating a subset of tasks, respectively. Examples include redirecting traffic and conducting end-to-end tests. Overall, Table 4 shows that operations other than task updates consume the majority of the time, emphasizing the importance of holistic optimization of the pipeline setup when deployment speed is a concern.

6.5.3 Real-world Example of Long Deploy Time

Large services that prioritize deployment safety may require multiple days to complete their `deploy` actions. To illustrate this, we present a real-world example of a foundational storage service at Meta that operates in every datacenter region. Its `deploy` action is configured to make progress on workdays from 9AM to 6PM without manual intervention. If the execution of the `deploy` action does not finish by 6PM, it will pause and continue on the next workday at 9AM.

Its deployment follows the *super linear* pattern for a few regions and then switches to the *region-by-region* pattern for the remaining regions. With over 20 phases in total, each of the early phases has a bake time of one hour, while the remaining phases have a bake time of 10 minutes. On average, task updates per phase take about 25 minutes. Taking into account the bake time, the early phases run for about 85 minutes each, while the remaining phases run for about 35 minutes each. Considering the total number of phases and their duration, the `deploy` action typically completes the early phases within one day and then resumes the next workday at 9AM. If everything goes smoothly, the deployment finishes on the second day. However, transient issues may cause it to retry and delay the completion time until the third day.

6.5.4 Recommendations for Pipeline Design

We recommend the following best practices for pipeline design. In general, we recommend using the *super linear* pattern as a starting point, as it provides a good balance between speed and safety. Moreover, we recommend including a phase in the middle of the pipeline to update all tasks within a region as opposed to never updating any whole region until the last phase. This is important because many services have regional dependencies, and certain issues, such as performance regressions, may only become noticeable when the code runs at the scale of a full region. Finally, we recommend scheduling deployments for the mornings of Monday to Thursday, so that developers have a full work day to troubleshoot any deployment issues. They may adopt continuous deployment after health checks and tests have matured.

7 Related work

There is a rich set of deployment tools, such as Spinnaker [42], AWS CodeDeploy [3], AWS CodePipeline [4], Azure Deployment Manager [46], Azure Pipeline [6], Google Cloud Build [17], Google Cloud Deploy [18], and CircleCI [13]. Cluster manager is also a well-studied topic. Examples include Kubernetes [22] and YARN [49] from open source, Borg [47, 50] from Google, and Protean [19] from Azure. Section 3 discussed advanced features that distinguish Conveyor and Twine from existing systems.

A number of prior works have studied and surveyed possible problems in software deployment [20, 23, 38, 41, 51], mostly based on open-source projects or individual case studies. As described in §3, the scale and diversity of the services at Meta have introduced many new challenges, such as in-place updates, handling complex code dependencies, fast deployment of large services like FrontFaaS, and deployment of complex ML models. Dedicated ML platforms such as AWS SageMaker can deploy models using the mirroring approach [35, 36], but they do not support in-place updates or the advanced model-deployment features described in §3.3.

Multiple works have focused on individual problems during deployment. For example, Gandalf tries to locate the problematic deployment after failures are detected [26]. ZebraConf tries to detect configuration updates that may cause compatibility issues [27]. Boyer et. al. [8] propose a declarative approach to update services, instead of the imperative approach used by most deployment tools, including Conveyor.

8 Conclusion

We presented the deployment scenarios, operational experience, and production data related to software deployment at Meta, along with the design and implementation of Conveyor. We demonstrated the feasibility of frequent and fully automated deployments supported by a single deployment tool for all services. Additionally, we presented novel techniques for in-place updates, analysis of code dependencies to prevent faulty releases, and the safe deployment of ML models.

Acknowledgments

This paper presents nine years of work by past and current members of several teams at Meta, including Conveyor, Twine, Health Check Service, Configuration Management, Release Engineering, and Inference Platform. In particular, we would like to call out the current members of the Conveyor team who are not on the author list: Alex Rock, Arvind Gautam, Brian Fitzpatrick, Eddy Li, Haydn Kennedy, Jared Bosco, Jimmy Zeng, Marcos Pertierra Arrojo, Marija Trifkovic, Matthew Boardman, Rudy Pikulik, Mike Belov, Matthew Almond, Nippun Goel, Shawn Cui, and Martin Reichhoff. We thank Hui Lei, all reviewers, and especially our shepherds, Ding Yuan and Malte Schwarzkopf, for their insightful comments.

References

[1] Release early, release often. https://en.wikipedia.org/wiki/Release_early,_release_often.

[2] Blue/Green Deployments. https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/bluegreen-deployments.html.

[3] AWS CodeDeploy. https://aws.amazon.com/codedeploy/.

[4] AWS CodePipeline. https://aws.amazon.com/codepipeline/.

[5] AWS Lambda. https://aws.amazon.com/lambda/.

[6] Azure Pipeline. https://azure.microsoft.com/en-us/products/devops/pipelines/.

[7] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[8] Fabienne Boyer, Nol de Palma, Xinxiu Tao, and Xavier Etchevers. A Declarative Approach for Updating Distributed Microservices. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 392–393, 2018.

[9] Nicolas Brousse. The Issue of Monorepo and Polyrepo In Large Enterprises. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, pages 1–4, 2019.

[10] Buck2. https://buck2.build/.

[11] Emily Burns, Asher Feldman, Rob Fletcher, Tomas Lin, Justin Reynolds, Chris Sanden, Lars Wander, and Rob Zienert. Continuous Delivery with Spinnaker. https://spinnaker.io/docs/concepts/ebook/.

[12] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.

[13] CircleCI. https://circleci.com/.

[14] eBPF. https://ebpf.io/.

[15] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, Carlsbad, CA, July 2022. USENIX Association.

[16] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.*, 30(4), November 2012.

[17] Google Cloud Build. https://cloud.google.com/build.

[18] Google Cloud Deploy. https://cloud.google.com/deploy.

[19] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 845–861. USENIX Association, 2020.

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

[21] 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.

[22] Kubernetes. https://kubernetes.io/.

[23] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, 82:55–79, 2017.

[24] 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 ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, 2021.

[25] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[26] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An Intelligent, End-to-End Analytics Service for Safe Deployment in Cloud-Scale Infrastructure. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 389–402, 2020.

[27] Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 410–425, 2021.

[28] Paul Marinescu. Autonomous testing of services at scale. https://engineering.fb.com/2021/10/20/developer-tools/autonomous-testing/, 2021.

[29] Caroline Moss. Facebook Went Down And People Started Calling The Cops, 2014. https://www.businessinsider.com/call-cops-when-facebook-is-down-2014-8.

[30] Guilherme Ottoni. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[31] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.*, 8(12):1816–1827, August 2015.

[32] Eric Raymond. The Cathedral and the Bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[33] Chuck Rossi. Rapid release at massive scale. https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/, 2017.

[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 Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.

[35] Amazon SageMaker. https://aws.amazon.com/pm/sagemaker.

[36] Amazon SageMaker UpdateEndpoint. https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_UpdateEndpoint.html.

[37] 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.

[38] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[39] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 111–120. IEEE, 2017.

[40] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017.

[41] Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar, and Liming Zhu. An Empirical Study of Architecting for Continuous Delivery and Deployment. *Empirical Software Engineering*, 24(3):1061–1108, 2019.

[42] Spinnaker. https://spinnaker.io/.

[43] Spinnaker rollout strategy for Kubernetes. https://spinnaker.io/docs/guides/user/kubernetes-v2/rollout-strategies/.

[44] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

[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] David Tepper. Introducing Azure Deployment Manager. https://learn.microsoft.com/en-us/archive/msdn-magazine/2019/august/azure-devops-introducing-azure-deployment-manager.

[47] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In

Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–14, 2020.

[48] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 805–816, 2015.

[49] 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.

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

[51] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306, 2018.