Turbine: Facebook's Service Management Platform for Stream Processing

Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, Guoqiang Jerry Chen
Facebook Inc.

***Abstract*—The demand for stream processing at Facebook has grown as services increasingly rely on real-time signals to speed up decisions and actions. Emerging real-time applications require strict Service Level Objectives (SLOs) with low downtime and processing lag—even in the presence of failures and load variability. Addressing this challenge at Facebook scale led to the development of *Turbine*, a management platform designed to bridge the gap between the capabilities of the existing general-purpose cluster management frameworks and Facebook's stream processing requirements. Specifically, Turbine features a fast and scalable task scheduler; an efficient predictive auto scaler; and an application update mechanism that provides fault-tolerance, atomicity, consistency, isolation and durability.**

Turbine has been in production for over three years, and one of the core technologies that enabled a booming growth of stream processing at Facebook. It is currently deployed on clusters spanning tens of thousands of machines, managing several thousands of streaming pipelines processing terabytes of data per second in real time. Our production experience has validated Turbine's effectiveness: its task scheduler evenly balances workload fluctuation across clusters; its auto scaler effectively and predictively handles unplanned load spikes; and the application update mechanism consistently and efficiently completes high scale updates within minutes. This paper describes the Turbine architecture, discusses the design choices behind it, and shares several case studies demonstrating Turbine capabilities in production.

***Index Terms*—Stream Processing, Cluster Management**

I. Introduction

The past decade has seen rapid growth of large-scale distributed stream processing in the industry. Several mature production systems have emerged including Millwheel [7], IBM Streams [16], Storm [29], Spark Streaming [35], Heron [21], StreamScope [23], Samza [26], Dataflow [8], and Flink [11]. These systems focus on a variety of challenges such as fault tolerance [29], low latency [11], [29], operability [21], [29], expressive programming model [7], processing semantics [8], [11], [26], scalability [7], effective resource provisioning [21], and state management [11], [23], [26], [35]. Similar trends are occurring at Facebook with many use cases adopting distributed stream processing for low latency analysis of site content interaction, search effectiveness signals, recommendation-related activities, and other uses. To satisfy these needs, we have built a framework which enables Facebook engineers to develop stream processing applications using declarative (SQL-like) and imperative (C++/Python/PHP) APIs [12]. A large number of stateless and stateful stream processing applications have been built [12], [18] using this framework. A requirement for the success of these applications is a scalable management platform to deploy, schedule, and re-configure these applications while maintaining their strict Service Level Objectives (SLOs) with low downtime and processing lag—even in the presence of failures and load variability. For example, many stream processing applications at Facebook require a 90-second end-to-end latency guarantee.

Existing general-purpose cluster management systems such as Aurora [1], Mesos [15], Borg [33], Tupperware [25] and Kubernetes [10] satisfy common management requirements across a variety of workloads but are not necessarily tuned for stream processing needs since they are designed based on different assumptions and use cases. Borg's [33] ecosystem consists of a heterogeneous collection of systems, and its users have to understand several different configuration languages and processes to configure and interact with the system. Kubernetes [10] builds upon Borg lessons by improving the experience of deploying and managing distributed services. YARN [30] is a resource manager adopted by many stream processing systems including Flink [11], Samza [26], Spark Streaming [35] and StreamScope [23]. They mostly provide resource isolation and enforcement framework which works best if resource requirements can be determined in advance. This is rarely the case for streaming workloads. Besides, they do not support auto scaling as a first-class citizen, which is the key to keep stream processing jobs real-time.

To bridge the gap between the capabilities of existing general-purpose cluster management systems and Facebook stream processing requirements, we have built *Turbine*, a service management platform that can be layered on top of the general-purpose frameworks listed above to provide specialized capabilities for stream processing applications. The current implementation of Turbine is integrated with Tupperware [25], Facebook's cluster management system similar to Google's Borg [33]. Tupperware is responsible for low-level host management while Turbine handles higher level job and task management to ensure that stream processing jobs continue to run with minimum downtime and meet their SLOs.

Turbine defines a proactive and preactive mechanism for automatically adjusting resource allocation in multiple dimensions (CPU, memory, disk and others). Elastic resource management plays an important role in maintaining streaming applications' SLOs in the presence of workload variations. To our knowledge, this is the first paper discussing details—including auto scaling results—of elastic resource manage-

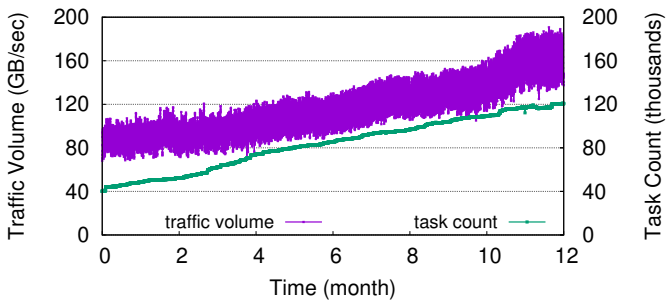


Fig. 1: The growth of Scuba Tailer service in a one-year interval, in terms of task count and input traffic volume.

ment in production involving large-scale clusters with several thousands of stream processing applications.

Another important component of Turbine is a fast scheduling mechanism for provisioning and managing stream processing tasks. Fast scheduling is a must in stream processing clusters since tasks are frequently moved, created, and deleted as a result of rebalancing and auto scaling (Turbine typically performs cluster-wide rebalancing every 15-30 minutes). Fast scheduling also benefits cluster-wide maintenance operations. Turbine, for instance, is capable of pushing a global stream-processing engine upgrade—an operation requiring a restart of tens of thousands of tasks—within 5 minutes.

Additionally, Turbine provides an *atomic*, *consistent*, *isolated*, *durable*, and *fault-tolerant* (ACIDF) application update mechanism. This capability is essential since multiple actors (provisioner service, auto scaler, human operator) may concurrently update the same job, and the system must ensure these updates are isolated and consistent. For example, changing the degree of parallelism and changing the input partition to task mapping for the same job must be serialized. To achieve that, Turbine automatically cleans up, rolls back, and retries failed job updates. Once an update is committed, it is guaranteed to be reflected in the cluster.

One of the main goals of Turbine is to provide high task availability. To help achieve this goal, Turbine architecture decouples *what to run* (Job Management), *where to run* (Task Management), and *how to run* (Resource Management). In case of individual Turbine component failures, this design allows the system to enter a degraded state in which stream processing tasks continue to run and process data.

Turbine has been running in production for over three years and is managing tens of thousands of diverse stream processing jobs on tens of thousands of machines. Turbine has been one of the core technologies that enabled a booming growth of stream processing at Facebook. As an example, Figure 1 shows the growth of one stream processing service powered by Turbine—Scuba Tailer (more details are in Section VI)—with doubled traffic volume during a one year period.

This paper makes the following contributions:

- An efficient predictive auto scaler that adjusts resource allocation in multiple dimensions;
- A scheduling and lifecycle management framework featuring fast task scheduling and failure recovery;
- An ACIDF application update mechanism;
- An architecture that decouples *what to run* (Job Management), *where to run* (Task Management), and *how to run* (Resource Management) to minimize the impact of individual component failures.

The rest of the paper is organized as follows. Section II provides an overview of Turbine and its components. Section III discusses Turbine's job management. Subsequently, Sections IV and V dive deeper into Turbine's task management and its elastic resource management capabilities. We share our production experience in Section VI and present lessons learned in Section VII. Section VIII discusses related work, and Section IX concludes.

II. System Overview

Figure 2 shows Turbine architecture. Application developers construct a data processing pipeline using Facebook's stream processing application framework, which supports APIs at both declarative level and imperative level. The complexity of the queries can vary from simple filtering and projection to a complex graph with multiple join operators or aggregations. After a query passes all validation checks (e.g., schema validation), it will be compiled to an internal representation (IR), optimized, then sent to the *Provision Service*. A query can be executed in batch mode and/or in streaming mode. The Provision Service is responsible for generating runtime configuration files and executables according to the selected mode. The batch mode is useful when processing historical data, and it uses systems and data from our Data Warehouse [28]. In this paper, we mainly focus on the stream processing part, represented in blue boxes in Figure 2.

Turbine contains components responsible for three major roles: job management, task management, and resource management. The Job Management layer stores job configurations and manages job updates. The Task Management layer converts job configurations to individual tasks, and then schedules and load-balances the tasks across clusters. The Resource Management layer is responsible for automatically adjusting resource allocation to a job, a task, and a cluster in real time. Turbine decouples the components that make decisions regarding *what to run* (decided in the Job Management layer), *where to run* (decided in the Task Management layer), and *how to run* (decided in the Resource Management layer). By doing so, Turbine can continue operating in a degraded mode even if any of these components fail, e.g., keep jobs running but not admitting new jobs.

A stream pipeline may contain multiple jobs, for example aggregation after data shuffling. A *Job* in turbine can have multiple *tasks* that run the same binary in parallel, with each task processing a disjoint subset of the input data. The binary implements a sequence of built-in and/or user-defined transformations, such as filtering, projection, aggregation, joins, data shuffling, and others. Turbine assumes a streaming data model as described in [12]. The communication between jobs is performed through Facebook's persistent message bus called Scribe instead of via direct network communication. Each

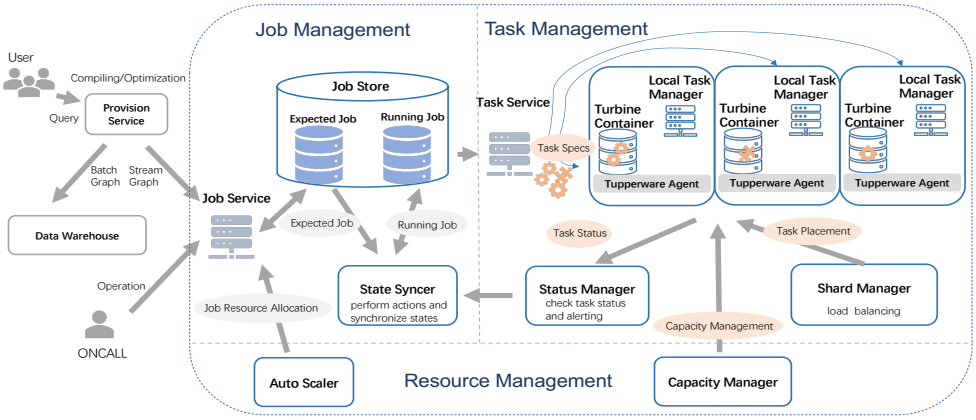


Fig. 2: Turbine's system architecture to manage stream processing services.

task of a job reads one or several disjoint data partitions from Scribe, maintains its own state and checkpoint, and writes to another set of Scribe partitions. Hence, a failed task can recover independently of other tasks by restoring its own state and resuming reading Scribe partitions from its own checkpoint. Turbine's data model eliminates dependencies between tasks. This reduces the complexity of recovering a failed task and makes task update/restart/move relatively lightweight. As a result, the logic of task scheduling, load balancing and auto scaling designed upon this data model is simplified, as it does not need to take task dependencies into consideration.

III. Job Management

A streaming application is compiled and optimized into a set of *jobs*, as aforementioned. After being provisioned, Turbine's Job Management layer is responsible for keeping all jobs up to date with respect to the configuration changes initiated by the user or by internal Turbine services. Turbine guarantees the changes to be performed in an ACIDF fashion. To this end, Turbine maintains 1) the *Job Store*—a repository containing the current and desired configuration parameters for each job, 2) the *Job Service*—a service to guarantee job changes are committed to the Job Store atomically, and 3) the *State Syncer*—a service that executes job update actions that drive jobs from the current state to the desired state. Configuration parameters cover all the information required for starting the tasks of a given job. Examples include names and versions of the binary, the number of tasks required for the job, and the resources allocated to the tasks.

ACIDF job updates play an important role to keep this complex system manageable and highly decoupled. In an environment with tens of thousands of jobs, job update failures and conflicts occur often as updates may come from many services. As a result, they must be resolved automatically and its results must be easy to reason about. Furthermore, job management should be flexible and extensible, as to accept new services if needed. Examples of new services include the auto scaler, which was added after Turbine was already in production, and an auto root-causer.

Given the considerations above, Turbine organizes job configurations in a hierarchical structure and separates what is expected to update from the actual action execution. As shown in Figure 2, the update is stored in the *Expected Job Configuration* table. The State Syncer executes and commits actions to the *Running Job Configuration* table only after actions are successfully performed.

A. Hierarchical Expected Job Configuration

In Turbine, a job can be updated by different internal services or users at the same time. Consider, for example, a job that is running with 10 tasks. The Auto Scaler decides to bump the number of tasks to 15. Meanwhile, Oncall1 and Oncall2 decide to bump the number of tasks to 20 and 30 respectively. Two problems need to be addressed here:

- **Isolation**: task count changes from the Auto Scaler, Oncall1 and Oncall2 cannot be performed at the same time, as changing job parallelism involves checkpoint re-distribution amongst tasks. If these actions are performed without isolation, checkpoints may end corrupted.
- **Consistency**: how to decide the final task count for the above job? A natural way is to serialize the task count changes by the time each action is issued, and to use the change from the last action. However, oncall interventions occurs when an automation service breaks, and we do not want oncall operations to be overwritten by a broken automation service.

Turbine addresses these problems using a hierarchical job configuration structure. The configuration management utilizes *Thrift* [24] to enforce compile-time type checking. This is then converted to a JSON representation using Thrift's JSON serialization protocol. Multiple configurations can be layered over each other, by *merging* the JSON configuration. We then employ a general JSON merging algorithm, that recursively traverses nested JSON structure while overriding values of the bottom layer with the top layer of configuration. This allows us to evolve the configuration specification, and layer an arbitrary number of configurations without changing the merging logic. The details are described in Algorithm 1.

Algorithm 1 Merge JSON Configs

```
1:  procedure LAYERCONFIGS(bottomConfig, topConfig)
2:      layeredConfig ← bottomConfig.copy()
3:      for each (key, topValue) ∈ topConfig do
4:          if isInstance(topValue, JsonMap) && (key in bottomConfig) then
5:              layeredValue ← layerConfigs(bottomConfig.get(key), topValue)
6:              layeredConfig.update(key, layeredValue)
7:          else
8:              layerConfig.update(key, topValue)
9:          end if
10:     end for
11:     return layeredConfig
12: end procedure
```

As illustrated in Table I, the *Expected Job Configuration* table contains four configuration levels. Each subsequent level takes precedence over all the preceding levels. The *Base Configuration* defines a collection of common settings—e.g., package name, version number, and checkpoint directory. The *Provisioner Configuration* is modified when users update applications. The *Scaler Configuration* is updated by the Auto Scaler whenever it decides to adjust a job's resource allocations based on the workload and current resource usage. The *Oncall Configuration* has the highest precedence and is used only in exceptional cases when human intervention is required to mitigate an ongoing service degradation.

The hierarchical design isolates job updates between different components: the Provision Service and the Auto Scaler modify respective configurations through the Job Service without needing to know anything about each other. The expected configurations are merged according to precedence, and provides a consistent view of expected job states. The Job Service also guarantees read-modify-write consistency when updating the same expected configuration, e.g., when different oncalls update oncall configurations for the same job simultaneously. Before a job update is written back to the expected job configuration, the write operation compares the version of the expected job configuration to make sure the configuration is the same version based on which the update decision is made. In addition, this hierarchical design is flexible since a new component can be added to the system by introducing a new configuration at the right level of precedence without affecting the existing components.

B. State Syncer

Turbine provides *atomic*, *durable* and *fault-tolerant* job updates by separating planned updates from actual updates. The planned updates are expressed in the *Expected Job Configurations* as discussed above, while the actual settings of the currently running jobs are stored in the *Running Job Configuration* table. The *State Syncer* performs synchronization between the expected and running job configurations every 30 seconds. In each round for every job, it merges all levels of the expected configurations according to their precedence, compares the result with the running job configurations, generates an *Execution Plan* if any difference is detected, and carries out the plan. An Execution Plan is an optimal sequence of idempotent actions whose goal is to transition the running job configuration to the expected job configuration. The State Syncer ensures:

TABLE I: Job store schema

Expected Job Table
Base Configuration
Provisioner Configuration
Scaler Configuration
Oncall Configuration

Running Job Table
Running Configuration

- **Atomicity** by committing to the running job configuration only after the plan is successfully executed.
- **Fault-tolerance** by aborting the failed plan and automatically re-scheduling the execution in the next round since differences between expected and running configurations are still detected.
- **Durability** by making sure the running job configurations are eventually synchronized with the expected job configurations, even when the State Syncer itself fails.

Performing synchronization operations in a cluster running tens of thousands of tasks, with tasks continuously moving hosts because of load balancing (see Section IV), is a highly choreographed and complex undertaking. To accelerate the synchronization, the State Syncer batches the *simple* synchronizations and parallelize the *complex* ones.

- *Simple synchronization* involves updates in which the Running Job Configuration is a direct copy of the merged Expected Job Configurations without further action required. Package release falls into this category: once the corresponding package setting is copied from the expected to the running job configurations, the package setting will eventually propagate to the impacted tasks (more details in Section IV). We can perform simple synchronizations of tens of thousands of jobs within seconds through batching.
- *Complex synchronization* is more than a single copy from expected to running job configurations. It usually involves multiple phases that must be coordinated and performed in a particular order (e.g., changing job parallelism) and may take a fairly long time to accomplish (e.g., initializing a new sink). Changing job parallelism, for instance, requires multi-step synchronizations: the State Syncer stops the old tasks first; once all the old tasks are completely stopped, it redistributes the checkpoints of the old tasks among the future new tasks, and only then starts the new tasks. If a complex synchronization fails in the middle, the State Syncer reschedules it in the next synchronization round within 30 seconds. If it fails for multiple times, the State Syncer quarantines the job and creates an alert for the oncall to investigate.

IV. Task Placement and Load Balancing

The goal of Turbine Task Management is to:

1) Schedule tasks without duplication, as at no point should the system have two active instances of the same task, even when other components of the Turbine system fail. There should also be no task loss. Only in scenarios where other components of the system fail, newly

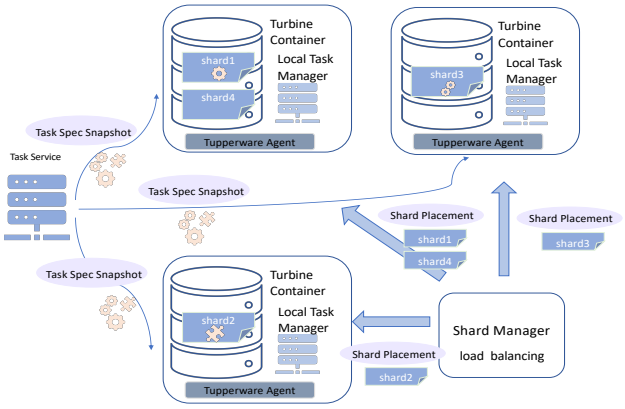


Fig. 3: Turbine's Task Management with a two-level distributed task scheduling and load balancing.

created tasks in the system might not be immediately scheduled.
2) Fail-over tasks to healthy hosts during host failures.
3) Restart tasks upon crashes.
4) Load balance tasks across the cluster for even CPU, Memory and IO usage.

Turbine integrates with Facebook's container manager (Tupperware) [25] and obtains an allocation of Linux containers. We call these the *Turbine Containers*. Each Turbine Container runs a local Task Manager that spawns a subset of stream processing tasks within that container. The mapping of tasks to Turbine Containers takes place in a distributed manner. Each local Task Manager periodically fetches the latest full snapshot of *Task Specs* using interfaces provided by the *Task Service*. It then updates the tasks that it is responsible for. A Task Spec includes all configurations necessary to run a task, such as package version, arguments, and number of threads. Internally, the *Task Service* retrieves the list of jobs from the Job Store and dynamically generates these task specs considering the job's parallelism level and by applying other template substitutions.

A. Scheduling

Turbine uses Facebook's Shard Manager service (similar to Google's Slicer [6]) which offers a general mechanism for balanced assignment of *shards* to containers. We discuss how tasks are physically scheduled to run in Turbine containers in this section, while the logic to decide task movement (load balancing) is discussed in the next section. Figure 3 shows an example of how this two-level placement works. A total of four shards are assigned to three Turbine containers. Each Task Manager gets a full snapshot of task specs from the Task Service and schedules the tasks belonging to the shards assigned to it. We explain some key aspects of this two-layer placement below.

1) Task to Shard Mapping: Each of the local Task Managers is responsible for determining which tasks are associated with the shards it hosts. Each of these Task Managers periodically (every 60 seconds) fetches the list of all Turbine tasks from the Task Service and computes an MD5 hash for each task. The result defines the shard ID associated with this task. This task to shard mapping is then cached in local Task Manager data structures. With every periodic fetch, this mapping is updated with any new tasks added to Turbine or for removing deleted tasks. Note also that by keeping the full list of tasks, we are able to perform load balancing and failover even if the Task Service is unavailable.

2) Shard Movement: When the Shard Manager decides to reshuffle assignments and move a shard from one Turbine container to another, it issues a `DROP_SHARD` request to the Task Manager running on the source container. The Task Manager stops the tasks associated with the dropped shard, removes the shard from its local bookkeeping structures, and returns `SUCCESS`. The Shard Manager then updates its mapping of shards to containers and sends an `ADD_SHARD` request to the Task Manager on the destination container. This Task Manager adds the new shard to its bookkeeping data structures, retrieves the list of tasks associated with the new shard, and starts them. To avoid excessive task downtime, if a `DROP_SHARD` or `ADD_SHARD` requests take too long, Turbine forcefully kills the corresponding tasks or initiates a Turbine container fail-over process respectively. The fail-over protocol is discussed in Section IV-C.

B. Load Balancing

Once the initial assignment of shards to Turbine containers is complete, Task Managers are responsible for starting the tasks corresponding to the shards assigned to their Turbine containers. As tasks run, Turbine periodically computes new shard load values which are then used by the Shard Manager to reshuffle the assignment of shards to Turbine containers.

Each Turbine container is associated with a capacity (specified in multiple dimensions (e.g., 26GB of memory). Each shard is associated with a load (i.e., resources it consumes in terms of CPU, memory, etc.). The algorithm to generate the shard to Task container mapping does a bin-packing of shards to Turbine containers such that the capacity constraint of each Turbine container is satisfied while also a global resource balance is maintained across the cluster. The resource balance is defined in terms of a utilization band per resource type. The algorithm does the mapping such that the total load of each Turbine container (calculated as the sum of the shard loads of the container) is within a band (e.g +/-10%) of the average of the Turbine container loads across the tier. In other words, the load difference between Turbine containers does not exceed an expected threshold. The algorithm also ensures additional constraints are satisfied, e.g., maintaining a head room per host, or satisfying regional constraints.

One important aspect to perform good load balancing is how to define shard load. Turbine provides different levels of resource guarantee by reporting different metrics. For example, for small C/C++ tasks, we report the dynamic resource usage of the tasks (e.g., average memory over the last 10 minutes). This helps Turbine to offset resource usage across tasks and improve cluster resource efficiency. On the other hand, for tasks using Java JVM or those that need cgroup enforcement,

we report the peak resource required (xmx, cgroup limits) to improve task stability.

A background load aggregator thread in each Task Manager collects the task resource usage metrics and aggregates them to calculate the latest shard load. This refreshed shard load is reported to the Shard Manager every ten minutes. The Shard Manager periodically (30 minutes for most of our tiers) re-generates the shard to Turbine container mapping using the latest shard load values. If this results in a new mapping, the Shard Manager moves shards according to the description in Section IV-A2.

C. Failure Handling

The main goal of the Turbine Failure Handling mechanism is to reduce the impact of system failures, ensure task availability, and guarantee that task management failures do not cause data corruption, loss, or duplication. Turbine uses a bi-directional heartbeat-based fail-over protocol with the Shard Manager. When the Shard Manager does not receive a heartbeat from a Task Manager for a full fail-over interval (default is 60 seconds), it assumes the corresponding Turbine container is dead, generates a new mapping for the shards in that container, and invokes the shard movement protocol described in Section IV-B and IV-A2.

However, sometimes, stopped heartbeats can be caused by other reasons such as connection failures. In this case, simply moving the lost shards to new containers may result in duplicate shards and eventually duplicated data processing. Turbine addresses this challenge by proactively timing out connections to the Shard Manager *before* the Shard Manager's fail-over interval period (in practice, timeout is configured to 40 seconds, fail-over is 60 seconds). If the connection times out, the Turbine container will reboot itself. After rebooting, if the container is able to re-connect to the Shard Manager before its fail-over, the shards would remain with the original container. Otherwise, Shard Manager would fail-over the shards from that container to new containers. The rebooted container will be treated as a newly added, empty container in that case. Shards will be gradually added to such containers as described in Section IV-A2.

D. Summary

To summarize, Turbine achieves fast scheduling and high task availability as follows:

- The two-level scheduling architecture decouples *what* and *where* to run; specifically, the Shard Manager does not need to interact with the Job Management layer. Such modular design simplifies debugging and maintenance of different system components and enables potential reuse of the (relatively general) load balancing and failure handling parts in other services.
- Each Task Manager has the full list of tasks, enabling Turbine to perform load balancing and fail-over even when the Task Service or Job Management layer are unavailable or degraded. If the Shard Manager becomes unavailable too, Turbine provides a further degraded mode where each Task Manager can fetch stored shard-container mapping.
- Each task manager has a local refresh thread to periodically (every 60 seconds) fetch from the Task Service. This guarantees task updates can be reflected in runtime after the Task Service caching expires (90 seconds) plus synchronization time in the State Syncer (refreshed every 30 seconds) mentioned in Section III. The overall end to end scheduling is 1-2 minutes on average, even for cluster-wide updates.
- If system failures occur, fail-overs start after 60 seconds. The downtime for a task on average is less than 2 minutes. This mechanism of handling failures also enables automatic handling of addition or removal of hosts in Turbine clusters, making Turbine elastic to use up all available resources.

V. Elastic Resource Management

Resource management is responsible for reacting to load changes at task level, job level and cluster level. It has two goals: (i) ensure that all jobs have sufficient resources needed for processing their input on time, and (ii) maintain efficient cluster-wide resource utilization. To achieve these goals, the resource management system must carefully weigh *when* and *how* resource allocation changes are made. Many systems [3], [8], [13], [19], [35] shed light on *when* to adjust the resource allocation. Turbine's *Auto Scaler* is built upon these approaches but meanwhile introduces a mechanism that not only adjusts resource effectively but also minimizes the amount of unnecessary churn in the system (e.g., restarting tasks that do not need to be restarted).

A. Reactive Auto Scaler

The first generation of the auto scaler was similar to Dhalion [13]. It consisted of a collection of *Symptom Detectors* and *Diagnosis Resolvers* and was purely reactive. This version focused on addressing the detection problem. It monitored pre-configured symptoms of misbehavior such as lag or backlog, imbalanced input, and tasks running out of memory (OOM). To measure lag, we define $time\_lagged$ for a job as:

$$time\_lagged = total\_bytes\_lagged / processing\_rate \quad (1)$$

where $total\_bytes\_lagged$ is the number of bytes available for reading that have not been ingested into the processing engine yet, and $processing\_rate$ is the number of bytes the streaming job can process per second. Intuitively, $time\_lagged$ corresponds to the amount of time the current processing is delayed from real time. Imbalanced input is measured by the standard deviation of processing rate across all the tasks belonging to the same job.

How to detect and measure OOMs depends on the per-task memory enforcement configuration. For tasks running in containers with configured cgroup limits, cgroup stats are preserved after OOM tasks are killed. Upon restarting such tasks, Turbine task managers read the preserved OOM stats and post them via the metric collection system to the Auto

Algorithm 2 Reactive Auto Scaler

```
for each job ∈ Jobs do
    if time lagged > SLO threshold then
        if imbalanced && job can be rebalanced then
            rebalance input traffic amongst tasks ∈ job
        else
            increase the number of task counts
        end if
    else if OOM then
        increase reserved memory
    else                          ▷ No OOM, no lag is detected in a day
        try to decrease resource assignment
    end if
end for
```

Scaler Symptom Detector. For Java tasks, JVM is configured to post OOM metrics right before it kills the offending tasks. For tasks without memory enforcement, Turbine task managers post ongoing memory usage metrics which are then compared by the Auto Scaler with the pre-configured soft memory limit to decide whether any memory adjustment is warranted. Algorithm 2 shows how the Reactive Auto Scaler works.

We encountered several problems with this reactive design: (i) it sometimes took too long for a single job to converge to a stable state due to lack of accurate estimation on required resources, as also observed in DS2 [19]; (ii) without knowing the lower bounds on the resource requirements for a given job, the Auto Scaler can make incorrect downscaling decisions causing a backlog in a previously healthy job; (iii) making scaling decisions without understanding the root cause of a particular symptom may amplify the original problem. For example, if a backlog is caused by excessive inter-service connection failures, increasing task parallelism may generate even more traffic for the dependent service, prolonging its unavailability even further. From our experience of running production stream processing applications, we have observed that the amount of resources needed for a given job is often predictable. Task footprints like maximal parsing rate are often stable as long as application logic and settings (e.g., checkpoint flush frequency) are unchanged. To incorporate this insight, we have built the second generation of the Auto Scaler by extending the original design with the ability to predict resource requirements proactively.

B. Proactive Auto Scaler

Resource Estimators and *Plan Generator* are introduced in this version (Figure 4). The purpose of a Resource Estimator is to estimate the usage of a given resource (e.g., CPU, memory, network bandwidth, and disk I/O) in a given job. The Plan Generator uses these estimates to construct a resource adjustment plan. Job resource estimation depends on the characteristic of the job. In typical Facebook workloads, jobs fall into two categories: stateless and stateful. Stateless jobs—filtering, projection, transformation—do not need to maintain state except for the checkpoints recording where to resume reading the input streams in case of task restarts. Stateful jobs—aggregation, join—maintain application-specific state in

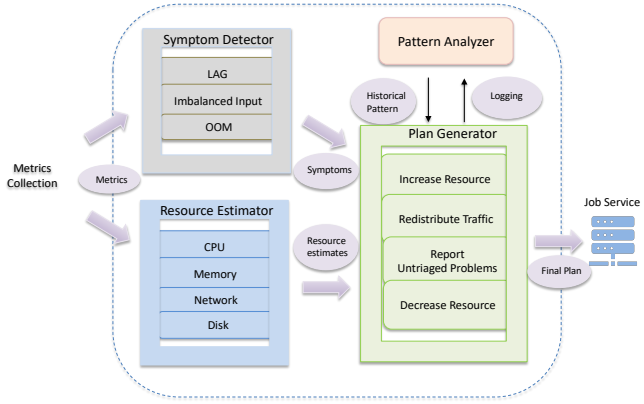


Fig. 4: The architecture of Turbine's Auto Scaler.

memory and persistent storage and must restore relevant parts of the state on restarts.

Stateless jobs are often CPU intensive and they perform input deserialization, data processing, and output serialization. Regardless which of these is dominant, CPU consumption is approximately proportional to the size of input and output data. For such jobs, input/output rate metrics are leveraged to estimate the maximum stable processing rate a single thread task can handle. The processing rate increases linearly with the number of tasks and threads in most cases, and the CPU resource unit needed for input rate X can be estimated as:

$$X/(P * k * n) \tag{2}$$

where P is the maximum stable processing rate for a single thread. Initially, P can be bootstrapped during the staging period (a pre-production phase for application correctness verification and performance profiling), and adjusted at runtime. Section V-C describes how to adjust P. Additionally, k is the number of threads per task, and n is the number of parallel tasks. If the backlogged data B needs to be recovered within time t, the CPU resource estimate is:

$$(X + B/t)/(P * k * n) \tag{3}$$

For stateful jobs, in addition to the CPU resource, memory and disk usage also needs to be estimated. For an aggregation job, the memory size is proportional to the key cardinality of the input data kept in memory. For a join operator, the memory/disk size is proportional to the join window size, the degree of input matching, and the degree of input disorder [18].

The *Resource Estimator* can be configured to estimate different dimensions of resource consumption and report them to the *Plan Generator* for further evaluation. The *Plan Generator* makes a synthesized decision based on symptoms and resource estimates collected. As an extension to Algorithm 2, the Plan Generator makes sure the final plan has enough resources to run a job based on the resource estimates from three aspects:

1) It prevents downscaling decisions from causing a healthy job to become unhealthy.
2) It ensures that untriaged problems (e.g., application bugs, network issues, dependency services) do not trig-

ger unnecessary and potentially harmful scaling decisions (see Section V-A).

3) It executes necessary adjustments for multiple resources in a correlated manner. For example, if a stateful job is bottlenecked on CPU, and the number of tasks is increased, the memory allocated to each task can be reduced.

C. Preactive Auto Scaler

When a scaling action is performed, it results in one or more tasks being added, removed, or restarted. This often involves CPU- and I/O-heavy initialization of the added and restarted tasks, which may adversely impact other tasks running on the same hosts. One main challenge in developing Auto Scaler is to ensure that it is sufficiently conservative not to cause unintended *cluster instability*—a state with a large portion of tasks being degraded.

To address this challenge, Turbine introduces the *Pattern Analyzer* whose goal is to infer patterns based on data seen and to apply this knowledge for pruning out potentially destabilizing scaling decisions. Turbine records and analyzes two sets of data:

1) **Resource Adjustment Data**. This information is used to adjust the footprint of a given job. For example, assuming that the current input rate is X, the current task count is n, and the max throughput per task is P, Turbine scaler plans to downscale the task count to $n' = ceiling(X/P)$ if no lag is detected for a long period of time. If this yields n' that is greater than n, our estimate of P must have been smaller than the actual max throughput. In this situation, Turbine adjusts P to the average task throughput and skips performing an action in this round. If $n' < n$, the scaler reduces the number of tasks in the job and observes the outcome. If an SLO violation is detected subsequently, the estimated value of P must have been greater than the actual max throughput and P needs to be adjusted to a value between X/n and P.
2) **Historical Workload Patterns**. Most stream processing jobs at Facebook exhibit diurnal load patterns: while the workload varies during a given day, it is normally similar—within 1% variation on aggregate—to the workload at the same time in prior days. These repeated patterns are leveraged to ensure that the scaler does not keep changing resource allocations too frequently. More specifically, Turbine records per minute workload metrics during the last 14 days, such as input rate or key set size for a job. When the scaler decides to change resource allocation, it verifies that this reduction will not cause another round of updates in the next x hours, for some configurable x, by checking historical workload metrics to validate that the reduced number of tasks was able to sustain traffic in the next x hours in the past. If the average input rate in the last 30 minutes is significantly different from the average of the same metric in the same time periods during the last 14 days, historical pattern-based decision making is disabled. We plan to address these outlier cases in our future work.

D. Untriaged Problems

Untriaged Problems are inevitable in reality. They are identified by misbehavior symptoms even when no imbalanced input is detected, and the job has enough resources according to the Auto Scaler estimates. These problems can be caused by many reasons including temporary hardware issues, bad user updates of the job logic, dependency failures, and system bugs. Hardware issues typically impact a single task of a misbehaving job; moving the task to another host usually resolves this class of problems. If a lag is caused by a recent user update, allocating more resources helps most of the time, and the job can converge to a stable state quickly after updated metrics are collected. Conversely, allocating more resources does not help in the case of dependency failures or system bugs. When Turbine cannot determine the cause of an untriaged problem, it fires operator alerts that require manual investigation.

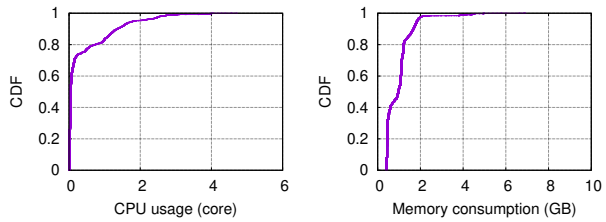
E. Vertical vs. Horizontal

The Turbine Auto Scaler supports *vertical* and *horizontal* scaling. Vertical scaling applies resource allocation changes within the task level without changing the number of tasks. Horizontal scaling involves changing the number of tasks to increase or decrease job parallelism. Horizontal scaling is challenging since changing the number of tasks requires redistributing input checkpoints between tasks for stateless jobs, and, additionally, redistributing state for stateful jobs. As discussed in Section III-B, such redistribution requires coordination between tasks and, as a result, takes more time. Conversely, vertical scaling may not always be desirable: large tasks make task movement and load balancing more difficult and they may cause resource fragmentation, making it difficult for the system to collocate other tasks in the same container. In Turbine, the upper limit of vertical scaling is set to a portion of resources available in a single container (typically $1/5$) to keep each task fine-grained enough to move. With this constraint, the Auto Scaler favors vertical scaling until this limit is reached, and only then applies horizontal scaling.

F. Capacity Management

The *Capacity Manager* monitors resource usage of jobs in a cluster and makes sure each resource type has sufficient allocation cluster-wide. It is authorized to temporarily transfer resources between different clusters for better global resource utilization. This is particularly useful during datacenter-wide events such as datacenter outages or disaster simulation drills. As described in Section IV-C, Turbine relies on automated fail-over to place tasks onto available healthy hosts. Hence, the procedure to add or remove hosts is fully automated.

When cluster-level resource usage spikes up—e.g., during disaster recovery—the Capacity Manager communicates with the Auto Scaler by sending it the amount of remaining resources in the cluster and instructing it to prioritize scaling



(a) (b)

Fig. 5: CPU and memory usage of Scuba Tailer tasks.

up privileged jobs. In the extreme case of the cluster running out of resources and becoming unstable, the Capacity Manager is authorized to stop lower priority jobs and redistribute their resources towards unblocking higher priority jobs faster.

VI. Production Experience

This section reports measurements from production Turbine clusters, evaluating resource utilization and response to workload change. We primarily focus on Scuba Tailer—the largest stream processing service managed by Turbine. Scuba [5] provides real-time, ad-hoc interactive analysis over arbitrary time-indexed datasets. It is often used in troubleshooting scenarios such as diagnosing performance regressions and analyzing the impact of infrastructure changes in near real time (within 90 seconds of the event occurrence). Ingestion of data into the Scuba backend is performed by Scuba tailers—stream processing jobs that read input data from Scribe, process it as needed, and send the result to the Scuba backend. Scribe is a persistent distributed messaging system offering high read and write throughput and low message delivery latency. At a logical level, Scribe data is partitioned into *categories* (c.f. Kafka topics [20]). Data for different Scuba tables is logged into different Scribe categories. For each Scuba table, there is a dedicated Scuba tailer job consisting of one or more tasks. Low latency and reliability are the main requirements.

The Scuba Tailer service uses a dedicated cluster of more than two thousand machines in three replicated regions. Each machine is equipped with 256GB memory, with a mix of 48 or 56 CPU cores. For each task, CPU overhead has a near-linear relationship with the traffic volume, while memory consumption is proportional to the average message size, since a tailer holds a few seconds worth of data in memory before processing and flushing it to the Scuba backend. Figure 5 shows the workload characteristics of over 120K tasks. Figure 5(a) reveals that over 80% of the tasks consume less than one CPU thread—an indication of low traffic. A small percentage of tasks need over four CPU threads to keep up with the incoming traffic. Memory consumption (Figure 5(b)) has a different distribution: every task consumes at least ∼400MB, regardless of the input traffic volume. The reason for this is that each task runs the Scribe Tailer binary as a subprocess and has an additional metric collection service. Overall, over 99% of the tasks consume less than 2GB.

A. Load Balancing

Turbine observes the resource consumption of all running tasks and spreads them to all available machines. Figure 6(a) and Figure 6(b) show the CPU and memory utilization in one cluster measured over one week. Figure 6(c) demonstrates that Turbine does a good job of distributing tasks across hosts—the number of tasks vary within a small range (from ∼150 to ∼230 per host). Note that to balance the load, Turbine only considers resource consumption, and not the number of tasks directly. The distribution shown in Figure 6 suggests that we could further increase resource utilization per host. We choose not to go this route and prefer to keep a certain headroom for absorbing workload spikes caused by changes in individual jobs, machine failures, and large-scale dependent services outages.

Before Turbine, each Scuba Tailer task ran in a separate Tupperware container. The migration to Turbine resulted in a ∼33% footprint reduction thanks to Turbine's better use of the fragmented resources within each container.

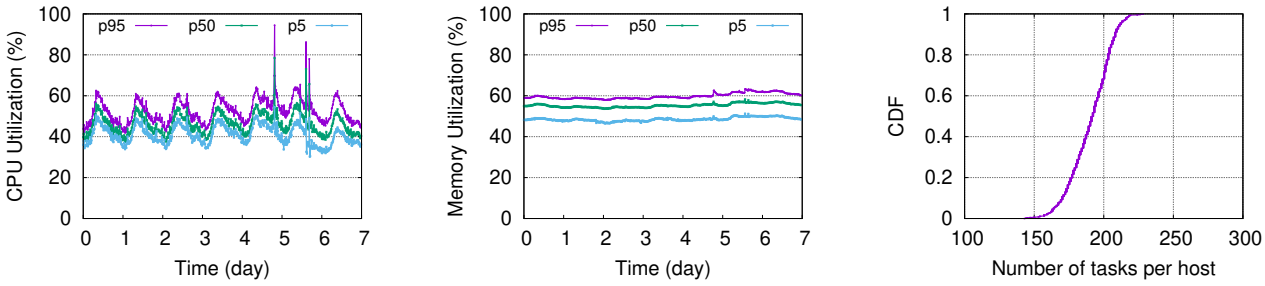
To demonstrate the effectiveness of Turbine's load balancing component, we conducted an experiment in a test cluster which shadows a fraction of production traffic. We first disabled the load balancer (hour 6), resulting in occasional spiky CPU utilization on some hosts. This was caused by traffic spikes in the input of some jobs and would have been mitigated by the load balancer moving other tasks off the busy hosts. To mimic maintenance events, we then manually triggered the failover on a few machines (hour 14), which resulted in imbalanced CPU and memory utilization across the cluster. For the jobs that were running on the machines with very high CPU utilization, we observed both lag due to lack of CPU, and crashes due to lack of memory. After we re-enabled the load balancer (hour 20), host resource consumption came back to normal levels very quickly. Figure 7 shows the changes in CPU utilization; memory utilization exhibits similar patterns (figures are omitted for brevity).

Proactive re-scheduling is instrumental in keeping the load balanced. Turbine is configured to periodically re-evaluate load metrics and redistribute tasks accordingly. Our measurement shows that each execution of the placement algorithm computing the mapping of 100K shards onto thousands of Turbine containers takes less than two seconds. We believe this is unlikely to become a scalability bottleneck even with 100× growth in the future. After the scheduling decision is made, the shard re-assignments are propagated to Task Managers.

B. Responding to Workload Change

Turbine automatically performs scaling actions to ensure that all jobs have sufficient resources and to maintain efficient cluster-wide resource utilization. This section describes several cases of the Auto Scaler in action as observed in production Turbine environments.

1) Scaling at Job Level: Figure 8 shows a Scuba tailer job that was disabled for five days due to application problems—resulting in a large backlog of data. When the application was fixed and re-enabled, it needed to process the backlog as



(a) (b) (c)

Fig. 6: In one Turbine cluster with > 600 hosts, CPU and memory utilization numbers are very close across hosts. With each host running hundreds of tasks, the reserved per-host headroom can tolerate simultaneous input traffic spike from many tasks.

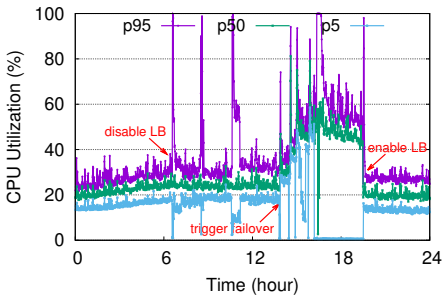


Fig. 7: Turbine's load balancer (LB) helps stabilize per-host resource utilization in situations like input traffic change and machines going offline.

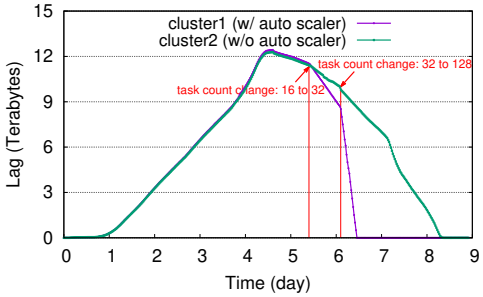


Fig. 8: Turbine's Auto Scaler helped a backlogged Scuba tailer job recover much faster.

quickly as possible. At the time, the Auto Scaler was available in `cluster1`, but it was not yet launched in `cluster2`. Figure 8 shows how the Auto Scaler helped with the recovery after the fixed application began processing the backlog. The purple line indicates that the Auto Scaler first scaled the job in `cluster1` to 32 tasks. 32 is the default upper limit for a job's task count for unprivileged Scuba tailers. It is used to prevent out of control jobs from grabbing too many cluster resources. After the operator temporally removed the limit, the Auto Scaler quickly scaled up to 128 tasks and redistributed the traffic to fully utilize the processing capability of each task. In comparison, the job in `cluster2` took more than two days (∼8× slower) to process the same amount of backlog. We manually increased the task count of the job in `cluster2` to 128 as well, but the recovery speed was still suboptimal because of uneven traffic distribution among tasks.

2) Scaling at Cluster Level: Facebook periodically practices disaster recovery drills, known as storms, that involve disconnecting an entire data center from the rest of the world [31]. During a storm, the traffic from the affected data center is redirected to other available data centers. The Turbine Auto Scaler plays an important role during a storm: it is responsible for scaling up jobs in the healthy data centers so they can handle additional redirected traffic. Figure 9 captures the aggregated task count change in a Turbine cluster (around 1000 jobs) as Turbine Auto Scaler was reacting to the storm traffic. The storm started on the morning of Day 2; the purple line shows that the cluster-wide traffic increased by ∼16% at peak time compared to the previous non-storm day. At the same time, the total task count was increased by ∼8% to handle the extra traffic. Recall that Turbine Auto Scaler performs vertical scaling first before attempting any horizontal scaling (Section V). This is why the task count change was relatively smaller compared to the traffic change. Before, during, and after the storm, ∼99.9% of jobs in that cluster stayed within their SLOs. After the storm ended, the total task count dropped to a normal level.

Figure 9 also subtly shows the effect of the predictive aspect of the Auto Scaler. Notice that the difference between the peak and low traffic during Day 1 is much larger than that between the peaks of Day 1 and Day 2. At the same time, the corresponding differences in task counts are not as pronounced. This is because the normal (Day 1) fluctuation is accounted for by the Auto Scaler's historical analysis as it tries to reduce task movements in anticipation of daily ebbs and flows (see Section V-C). Conversely, the relatively higher traffic during Day 2 peak is unexpected, and, so, the Auto Scaler is forced to add a relatively high number of tasks.

3) Scaling for Resource Efficiency: Turbine Auto scaler improves resource efficiency as well as helps keeping jobs within their SLOs. Without auto scaling, jobs have to be over-provisioned to handle peak traffic and reserve some headroom for unexpected traffic spikes. Figure 10 records the CPU and memory footprints when auto scaling was launched in one Scuba Tailer cluster. The overall task count dropped from ∼120K to ∼43K, saving ∼22% of CPU and ∼51% of memory. After the rollout, the Capacity Manager was authorized to reclaim the saved capacity.

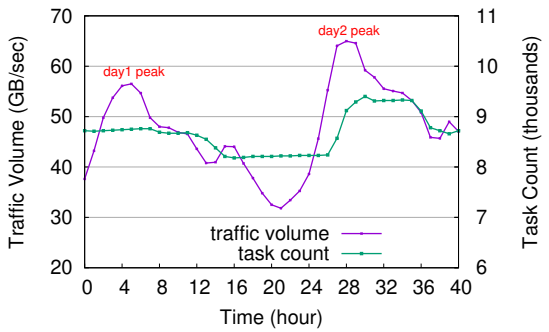


Fig. 9: Turbine's Auto Scaler reacted to Storm by performing horizontal scaling at cluster level.

VII. Lessons Learned

Facebook's streaming workload is highly variable. This is due to diurnal peak, quick experimentations, or even application bugs from upstream. Unlike batch pipelines that process offline on hourly or daily partitions and can tolerate a certain degree of delay by letting the processing wait for resources, stream processing must maintain low latency. Successful systems quickly grow too large for a human to be involved in handling workload changes. The importance of this lesson only increases with growth: any feature that involves manual effort to scale eventually becomes an operational bottleneck.

A significant part of large-scale distributed systems is about *operations at scale*: scalable monitoring, alerting, and diagnosis. Aside from job level monitoring and alert dashboards, Turbine has several tools to report the percentage of tasks not running, lagging, or unhealthy. For each of these higher level metrics, we have a comprehensive runbook, dashboards, and tools that drill down into the root cause of the problem. Our comprehensive monitoring, diagnosis, self-adjusting services, and the Facebook ecosystem of monitoring tools helps to keep the clusters healthy.

VIII. Related Work

Cluster management systems like Tupperware [25], Borg [33], Omega [27], Kubernetes [10], and Mesos [15] are all examples that enable co-location of latency-sensitive and batch jobs in shared clusters. Turbine is a nested container infrastructure built on top of Tupperware: the Turbine Container serves as the parent container managing a pool of resources on each physical host. Stream processing tasks are run as children containers below the Turbine Container. Building Turbine on top of a cluster management system rather than simply using it directly allowed us to be more focused on streaming-specific needs: fast scheduling, load balancing, and auto scaling.

YARN [30] and Corona [4] are resource managers for big data analytics workloads initially built for batch applications. They provide resource isolation and enforcement frameworks which work best if resource requirements can be determined in advance. This is rarely the case for streaming workloads. Though Turbine has been tailored for supporting stream processing jobs at Facebook scale, we have been investigating the integration of Turbine with the YARN-like resource manager used in Facebook's Data Warehouse so we can co-locate batch and streaming jobs in the future. We believe that the end result will retain the essential characteristics of Turbine enabling fast scheduling and resource reconfiguration.

Job scheduling has been studied extensively. This topic covers centralized [14], [32], [34] and distributed [9], [27] scheduling frameworks. An important characteristic of a distributed scheduler is how it deals with scheduling conflicts. Omega [27] adopts optimistic concurrency control [22]: if a scheduling request fails, it will be rolled back and retried later. Apollo [9] postpones any corrective action until tasks are dispatched and the on-node task queue can be inspected for resource availability. Both of these approaches introduce a potential scheduling delay. In stream processing, we prefer to schedule jobs as quickly as possible. Turbine focuses less on initial scheduling but relies on subsequent load balancing to mitigate poor scheduling choices. In practice, this works well because load balancing is quick and can be further optimized by packing more tasks into one shard without increasing the total number of shards.

Declaration-based resource management approaches [2], [14], [17], [34] work by allocating resources according to each job's specified amount or relative shares. In these systems, fairness is often one optimization goal, with the assumption that some jobs can be throttled when resources are low. Turbine takes a different approach: it allocates resources based on each job's observed consumption and proactively balances the load to avoid hot hosts. As a result, Turbine is able to adopt a simpler resource allocation algorithm and meanwhile achieve good resource utilization. Turbine throttles resource consumption by stopping tasks only as a last resort, and does so by prioritizing the availability of tasks belonging to high business value applications.

IX. Conclusion and Future Work

We have described Turbine, a large-scale management framework for stream processing applications. Turbine leverages a mature cluster management system and enhances it with loosely coupled microservices responsible for answering *what to run* (Job Management), *where to run* (Task Management), and *how to run* (Resource Management)—yielding a high scale and high resiliency management infrastructure capable of supporting a large number of pipelines processing a large amount of data with little human oversight.

Going forward, we plan to investigate machine learning techniques for automatic root cause analysis and mitigation of incidents that previously required human intervention. Additional avenues of exploration include maximizing resource utilization by reducing the reserved capacity headroom and optimizing task placement with the help of a continuous resource estimation algorithm.

X. Acknowledgments

We would like to thank all those who contributed to Turbine including Ankur Goenka, Xiangyu Ji , Anshul Jaiswal, Albert Kim, Wenbang Wang, Matt Dordal, Li Fang, Alex Levchuk,

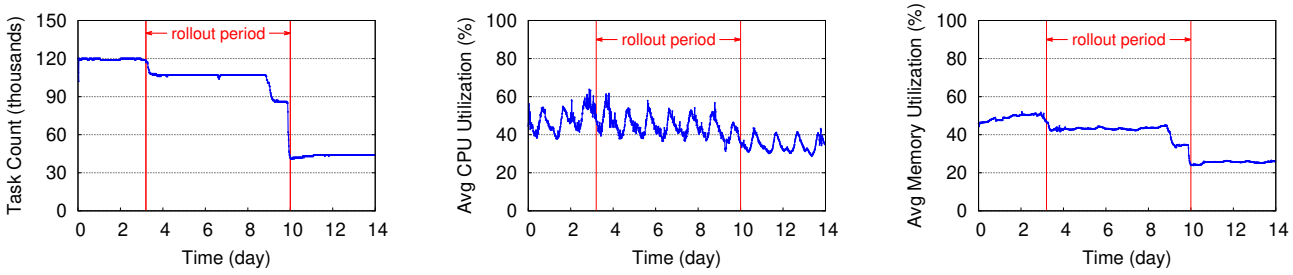


Fig. 10: The production results of slowly rolling out the auto scaling capability to one Scuba Tailer cluster.

Mark Crossen, Yichen Yao, Shuo Xu, Zhou Tan, George Xue, Konstantin Evchenko, Jim Nisbet. We are also indebted to Rajesh Nishtala, Sriram Rao and anonymous reviewers for reviewing the paper and insightful comments.

REFERENCES

[1] Apache Aurora. http://aurora.apache.org/.

[2] Hadoop Capacity Scheduler. https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.

[3] No shard left behind: dynamic work rebalancing in Google Cloud Dataflow. https://cloud.google.com/blog/products/gcp/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow.

[4] Under the Hood: Scheduling MapReduce jobs more efficiently with Corona. https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920/.

[5] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at Facebook. *VLDB*, 2013.

[6] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *OSDI*, 2016.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *VLDB*, 2013.

[8] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB*, 2015.

[9] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):10:70–10:93, Jan. 2016.

[11] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *VLDB*, 2017.

[12] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at Facebook. In *SIGMOD*, 2016.

[13] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in Heron. *VLDB*, 2017.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[16] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM J. Res. Dev.*, 57(3-4):1:7–1:7, May 2013.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[18] G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, A. Banerjee, B. Heintz, S. Iyer, and A. Jaiswal. Providing streaming joins as a service at Facebook. *VLDB*, 2018.

[19] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*, 2018.

[20] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB*, 2011.

[21] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *SIGMOD*, 2015.

[22] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[23] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[24] A. A. Mark Slee and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. https://thrift.apache.org/static/files/thrift-20070401.pdf, 2007.

[25] A. Narayanan. Tupperware: Containerized deployment at Facebook. In *DockerCon*, 2014.

[26] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *VLDB*, 2017.

[27] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[28] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[29] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[30] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[31] K. Veeraraghavan, J. Meza, S. Michelson, S. Panneerselvam, A. Gyori, D. Chou, S. Margulis, D. Obenshain, S. Padmanabha, A. Shah, Y. J. Song, and T. Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *OSDI*, 2018.

[32] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[33] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[34] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[35] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.