# Dhalion: Self-Regulating Stream Processing in Heron

Avrilia Floratou
Microsoft
avflor@microsoft.com

Ashvin Agrawal
Microsoft
asagr@microsoft.com

Bill Graham
Twitter, Inc.
billg@twitter.com

Sriram Rao
Microsoft
sriramra@microsoft.com

Karthik Ramasamy
Streamlio
karthik@streaml.io

## ABSTRACT

In recent years, there has been an explosion of large-scale real-time analytics needs and a plethora of streaming systems have been developed to support such applications. These systems are able to continue stream processing even when faced with hardware and software failures. However, these systems do not address some crucial challenges facing their operators: the manual, time-consuming and error-prone tasks of tuning various configuration knobs to achieve service level objectives (SLO) as well as the maintenance of SLOs in the face of sudden, unpredictable load variation and hardware or software performance degradation.

In this paper, we introduce the notion of *self-regulating* streaming systems and the key properties that they must satisfy. We then present the design and evaluation of Dhalion, a system that provides self-regulation capabilities to underlying streaming systems. We describe our implementation of the Dhalion framework on top of Twitter Heron, as well as a number of policies that automatically reconfigure Heron topologies to meet throughput SLOs, scaling resource consumption up and down as needed. We experimentally evaluate our Dhalion policies in a cloud environment and demonstrate their effectiveness. We are in the process of open-sourcing our Dhalion policies as part of the Heron project.

## 1. INTRODUCTION

In a world where organizations are being inundated with data from internal and external sources, analyzing data and reacting to changes in real-time has become a key service differentiator. Examples for such needs abound - analyzing tweets to detect trending topics within minutes, reacting to news events as soon as they occur, as well as surfacing system failures to data center operators before they cascade.

The ubiquity of these use cases has led to a plethora of distributed stream processing systems being developed and deployed at data center scale in recent years (see Apache Storm [26], Spark Streaming [10] and Twitter's Heron [22], LinkedIn's Samza [4], etc). Given the scales at which these systems are commonly deployed, they are naturally designed to tolerate system failures and coexist with other applications in the same clusters.

*Proceedings of the VLDB Endowment,* Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

However, a crucial challenge that has largely escaped the attention of researchers and system developers is the complexity of configuring, managing and deploying such applications. Conversations with users of these frameworks suggest that these manual operational tasks are not only tedious and time-consuming, but also error-prone. Operators must carefully tune these systems to balance competing objectives such as resource utilization and performance (throughput or latency). At the same time, they must also account for large and unpredictable load spikes during provisioning, and be on call to react to failures and service degradations.

Motivated by these challenges, in this paper we present Dhalion[1], a system that is built on the core philosophy that stream processing systems must **self-regulate**. Inspired by similar notions in complex biological and social systems, we define three important capabilities that make a system self-regulating.

**Self-tuning:** In the quest to support diverse applications and operational environments, modern stream processing systems are typically highly configurable, exposing various knobs that allow operators to tune the system to their particular needs. However, this feature is both a boon and bane. Users of streaming frameworks frequently complain about the amount of manual effort that is required even for the simplest tasks, such as determining the degree of parallelism at each stage of a streaming pipeline. Since there is no principled way to fully determine the ideal configuration, users typically try several configurations and pick the one that best matches their service level objectives (SLO). A self-regulating streaming system should take the specification of a streaming application as well as a policy defining the objective, and automatically tune configuration parameters to achieve the stated objective.

**Self-stabilizing:** Long-running streaming applications are inevitably faced with various external shocks that can threaten their stability. For instance, it is common for Twitter's tweet processing applications to experience loads that are several times higher than normal when users all over the world react to earthquakes, celebrity faux pas or World Cup goals. Since these load variations are largely unpredictable, operators are forced to over-provision resources for these applications to avoid SLO violations. But this choice implies that in most cases, the system resources are underutilized, thus lowering the cluster operator's return on investment (ROI). A self-regulating streaming system must react to external shocks by appropriately reconfiguring itself to guarantee stability (and SLO adherence) at all times. It may satisfy this requirement by acquiring additional resources and scaling up under load spikes, and thereafter relinquishing resources and scaling back down after load stabilizes.

[1]In Greek mythology, Dhalion is a bird with magical healing capabilities.

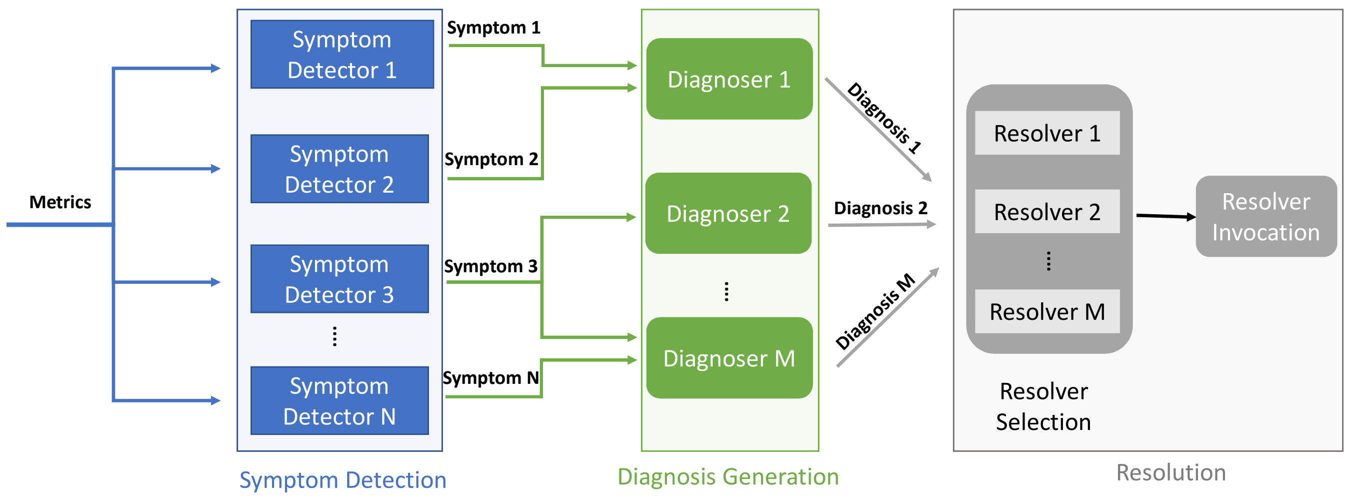

**Figure 1: Dhalion policy phases**

**Self-healing:** Most streaming systems incorporate various fault-tolerant mechanisms to recover from hardware or software failures. However, system performance can be affected not only by failures but also by hardware and software delivering degraded quality of service. For example, it is possible for a machine in a cluster to be nominally available, while delivering abysmally low performance either due to hardware issues (such as a slow disk) or software issues (such as memory constraints leading to swap thrashing). A self-regulating streaming system must identify such service degradations, diagnose the internal faults that are at their root, and perform the necessary actions to recover from them.

Dhalion is a system that essentially allows stream processing frameworks to become self-regulating. Dhalion has been implemented and evaluated on top of Twitter Heron [22] and we are in the process of releasing it to open-source as a contribution to the Heron code base. However, its architecture and basic abstractions are also applicable to other streaming engines as well.

Dhalion sits on top of each Heron application and periodically invokes a well-specified policy. The policy examines the status of the streaming application and detects potential problems, such as existence of slow processes, lack of resources, SLO violations etc. After diagnosing a particular problem, the Dhalion policy attempts to resolve it by performing the appropriate actions. An important aspect of Dhalion is its extensible and modular architecture. Dhalion is flexible enough to incorporate new policies that users can implement using well-specified APIs. Moreover, since the policies employ several self-contained modules, Dhalion users can reuse existing modules when specifying their own policies.

Motivated by the challenges that users of streaming systems face, we designed two Dhalion policies, namely dynamic resource provisioning for throughput maximization in the presence of load variations, and auto-tuning of the Heron application for meeting throughput SLOs. The first policy provides Heron with self-stabilizing and self-healing capabilities. The second policy additionally provides self-tuning capabilities. To the best of our knowledge, none of the existing streaming systems employs such sophisticated policies, but they mostly rely on user intervention.

In this work, we make the following contributions:

1. Motivated by challenges that users face, we introduce the notion of self-regulating streaming systems and discuss their major properties.
2. We design Dhalion, a novel system that sits on top of streaming engines and allows them to become self-regulating through the invocation of well-specified Dhalion policies (Section 2). Dhalion has a modular and extensible architecture and has been implemented on top of Twitter Heron (Section 4).
3. We present two important Dhalion policies that allow Heron to dynamically provision resources in the presence of load variations and to auto-tune Heron applications so that a throughput SLO is met (Section 5).
4. We evaluate our policies on top of Heron in a cloud environment and demonstrate the effectiveness of Dhalion (Section 6).

We discuss related work in Section 7. Finally, we conclude by outlining directions for future work (Section 8). In what follows, we start by presenting a high-level overview of Dhalion with an emphasis on its key abstractions.

## 2. DHALION OVERVIEW

In this section, we present a high-level overview of Dhalion with an emphasis on its key abstractions. In the following section, we discuss Dhalion's architecture in detail.

Dhalion sits on top of a streaming engine such as Heron, and provides self-regulating capabilities to it through a modular and extensible architecture. Dhalion periodically invokes a *policy* which evaluates the status of the topology, identifies potential problems and takes appropriate actions to resolve them. Users and system administrators can define new policies based on their particular needs and workload requirements. For example, a user might create a policy that attempts to maximize throughput without overprovisioning resources or a policy that automatically provisions necessary resources in order to meet a particular Service Level Objective (SLO). Figure 1 presents the various phases of a Dhalion policy. These phases will be discussed in detail in Section 4.

In the `Symptom Detection` phase, Dhalion observes the system state by collecting various metrics from the underlying streaming system. Some example metrics are the rate at which tuples are processed at a particular topology stage or the number of packets pending for processing at a particular task. Based on the metrics collected, Dhalion attempts to identify *symptoms* that can potentially denote that the health of the streaming application has been compromised. For example, a Heron instance uses *backpressure* as a mechanism to notify its upstream sources that is unable to keep up with its input rate and requires its sources to slow down. Accordingly, Dhalion identifies backpressure as a symptom that shows that the streaming pipeline is not currently in a healthy state. Another symptom is processing skew across the tasks of a particular stage in the pipeline. If some tasks of a given stage process significantly more tuples than the remaining instances of the same stage, then this symptom is worth investigating further.

After collecting various symptoms, in the `Diagnosis Generation Phase`, Dhalion attempts to find one or more *diagnoses* that explain them. For example, the existence of backpressure can be attributed to various reasons such as resource underprovisioning at a particular stage, slow hosts/machines or data skew. Dhalion produces all the possible diagnoses that can explain the observed symptoms.

Once a set of diagnoses has been found, the system evaluates them and explores the possible actions that can be taken to resolve the problem during the `Resolution Phase`. For example, in case Dhalion has diagnosed that backpressure is caused because of the limited resources assigned to a specific stage, then to resolve the issue, it can scale up the resources assigned to this stage. Similarly, if backpressure is created because of one or more tasks are running slower due to a slow machine, then Dhalion can potentially resolve this issue by moving the tasks to a new location.

Note that it is possible that a diagnosis produced by Dhalion is erroneous and thus, an incorrect action is performed that will not eventually resolve the problem. For instance, Dhalion might diagnose that backpressure is caused because of limited resources assigned to the tasks of a given stage, whereas it is actually due to a slow task. This can happen for example, when the task is not significantly slower than its peers and thus Dhalion didn't classify it as an outlier. In this case, the system will incorrectly scale up the topology resources. For this reason, after every action is performed, Dhalion evaluates whether the action was able to resolve the problem or brought the system to a healthier state. If an action does not produce the expected outcome then it is *blacklisted* and it is not repeated again. This mechanism is very powerful since it allows the system to avoid repetition of erroneous actions. In the absence of a blacklist mechanism, it is possible that the system can get stuck in an unhealthy situation and fall into the trap of repeatedly invoking fruitless (maybe even counter-productive) actions.

## 3. HERON BACKGROUND

We have implemented Dhalion by extending Twitter's Heron [7] system thereby providing self-regulating capabilities to Heron. Before describing our implementation, we present a brief overview of Heron and its rate control mechanisms. An extensive discussion on Heron's architecture can be found in [17, 22].

### 3.1 Topology Architecture

Heron users deploy topologies which are essentially directed graphs of spouts and bolts. The spouts are sources of input data such as streams of tweets, whereas the bolts represent computations on the streams they receive from spouts or other bolts. Spouts often read data from queues, such as Kafka [3] or Distributed Log [6] and generate a stream of tuples, which is in turn consumed by a network of bolts that perform the actual computations on the stream. Figure 2 shows the architecture of a topology. In this section, we focus on the gray components of the figure which depict the most important Heron components of the topology. Dhalion (in blue), which is implemented on top of Heron, will be extensively discussed in the following sections.

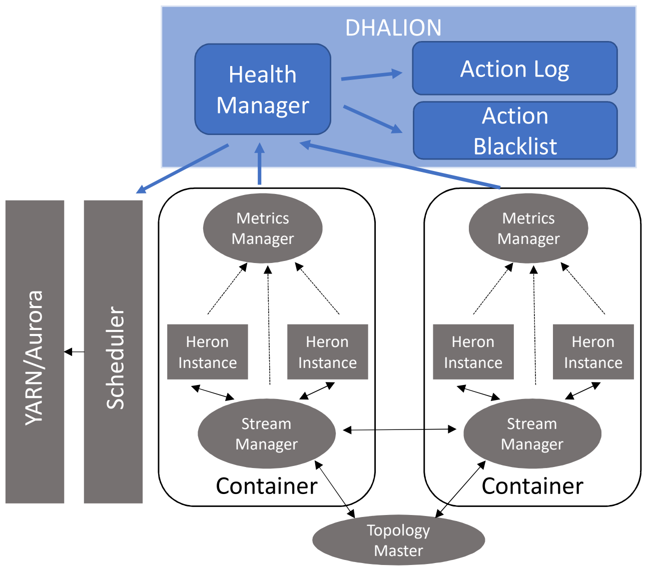

**Figure 2: Topology architecture**

Heron operates on top of a scheduling framework such as Aurora [1] or YARN [27]. Each Heron topology is deployed on containers managed by these frameworks. The `Scheduler` component is responsible for requesting containers from the underlying scheduling framework. The `Topology Master` process is responsible for managing the topology throughout its existence and occupies one container. The remaining containers each run a `Stream Manager`, a `Metrics Manager`, and a number of processes called `Heron Instances` which run the code that corresponds to the user logic. Each spout/bolt is represented by a set of `Heron Instances` that independently and in parallel, execute the user code that corresponds to this spout/bolt. The `Stream Manager` is a critical component of the system as it manages the routing of tuples among `Heron Instances`. More specifically, each `Heron Instance` connects to its local `Stream Manager` to send and receive tuples. All the `Stream Managers` in a topology connect between themselves to form $n^2$ connections, where $n$ is the number of containers of the topology. Finally, the `Metrics Manager` is responsible for collecting and reporting various metrics from the `Stream Manager` and `Heron Instances` located in the same container.

### 3.2 Topology Backpressure

An important aspect of Heron is its rate control mechanism. Rate control is crucial in topologies where different components can execute at different speeds or where the processing speed of the components can vary over time. This can happen due to various reasons such as limited number of `Heron Instances` in one or more topology stages (limited parallelism), data skew or because of slow machines/containers. Heron dynamically adjusts the rate at which data flows through the topology using a backpressure mechanism. As an example, consider a topology in which the downstream stages are slow due to one of the reasons mentioned previously. If the upstream topology stages do not reduce the rate at which they emit data, tuples will be accumulated in long queues and as a result the system might start dropping tuples. Heron's backpressure mechanism slows down the upstream stages so that such situations are avoided. As we will discuss in Section 5, Dhalion treats the existence of topology backpressure as an indication that the system is unstable and therefore, takes appropriate actions to bring the topology back to a healthy state.

The Heron Stream Manager has a significant role in handling and propagating backpressure across the topology stages. The backpressure mechanism works as follows: When one or more Heron Instances in a container is slower than its peers, the local Stream Manager recognizes that event as its buffer that keeps the tuples to send will fill up. The Stream Manager then stops reading data from the local spouts and sends special messages to the other Stream Managers requesting them to stop reading data from the local spouts as well. Once the slow Heron Instances catch up, the local Stream Manager notifies the other Stream Managers and they in turn start reading data from their local spouts again.

## 4. DHALION ARCHITECTURE

In this section, we describe Dhalion's architecture in detail. As shown in Figure 2, Dhalion consists of three major components, namely the Health Manager, the Action Log and the Action Blacklist. In the following sections, we discuss these three components in detail.

### 4.1 Health Manager

The Health Manager is a critical component of Dhalion as it is responsible for maintaining the health of the running topology. The Health Manager is a long running process that periodically invokes a policy which evaluates the status of the topology, identifies potential problems and takes appropriate actions to resolve them. Dhalion supports two kinds of policies: *invasive* and *non-invasive*. An *invasive* policy takes actions that adjust the topology configuration (e.g, parallelism changes). A *noninvasive* policy on the other hand, does not make any topology changes but typically alerts the user when a particular event takes place. The Health Manager can execute multiple *noninvasive* policies concurrently but only one *invasive* policy at a time. This is because executing multiple *invasive* policies concurrently might result in conflicting actions and that can cause system instability.

As we will discuss in Section 5, Dhalion already implements various policies. However, Dhalion's Health Manager supports an extensible set of policies by defining a policy API. This allows users to create new *invasive* and *noninvasive* policies that can be invoked by the Health Manager. Figure 1 shows the various phases of a policy. As shown in the figure, the policy evaluation consists of three major phases that we describe below:

**Symptom Detection Phase**: During this phase, Dhalion collects several metrics from the Metrics Managers and attempts to identify symptoms that could potentially denote that the health of the topology has been compromised (such as backpressure, skew, etc.). The identification of symptoms is done through various Symptom Detectors that collect the appropriate metrics and perform the necessary metric evaluations. The Symptom Detectors employ a variety of anomaly detection techniques such as outlier detection and clustering of data points among others. Once a symptom is identified, the Symptom Detector produces a *symptom description* which is a compact representation of the symptom along with the metrics and their corresponding values used to identify this symptom. For example, once the Backpressure Detector detects the existence of backpressure, it produces a description of the symptom specifying which bolt in the topology initiated the backpressure and how much time the Heron Instances corresponding to this bolt suspended the input data consumption. Dhalion implements various Symptom Detectors but also provides well-specified APIs so that users can create their own Symptom Detectors and incorporate them to their policies.

**Diagnosis Generation Phase**: The Diagnosis Generation Phase collects various symptoms produced by the Symptom Detection Phase and attempts to identify the root cause of these symptoms. This is accomplished through various Diagnosers, which are modules designed to get as input a set of symptom descriptions and produce a *diagnosis* based on these symptoms if possible. For example, as we discuss later, the Resource Underprovisioning Diagnoser takes as input a backpressure symptom description and determines whether the existence of backpressure can be attributed to the small number of Heron Instances in a particular stage of the topology (resource underprovisioning). Similarly, the Slow Instances Diagnoser determines whether the cause of backpressure can be one or more instances running slower than their peers in a particular topology stage because one or more containers or machines might be slow.

The Diagnosers produce a *diagnosis description*, which is a succinct representation of the root cause of the problem along with the symptoms and their corresponding metric values that led to this specific diagnosis. The current set of Diagnosers implemented in Dhalion make a binary decision essentially determining whether the symptoms can be attributed to a particular cause or not. However, users can also create Diagnosers that assign a confidence level to the diagnosis that they produce if needed. Finally, similar to the Symptom Detectors, the Diagnosers have a well-specified API that allows the users to create new Diagnosers and incorporate them to their policies.

**Resolution Phase**: The Resolution Phase is the last phase of the policy. Its major goal is to resolve the problems identified by the Diagnosis Generation Phase by taking the necessary actions. The basic building block of this phase is the Resolver module. A Resolver takes as input a diagnosis description and based on that, performs the appropriate action to bring the topology back to a healthy state. There is typically a 1-1 mapping between Diagnosers and Resolvers. For example, the Scale Up Resolver can solve problems identified by the Resource Underprovisioning Diagnoser by increasing the number of Heron Instances that correspond to the topology stage that initiates the backpressure. Similarly, the Restart Instances Resolver moves the Heron Instances that have been identified as slow by the Slow Instances Diagnoser to new containers. In Section 5, we will extensively discuss the functionality of these components. Similar to Symptom Detectors and Diagnosers, users have the flexibility to incorporate new Resolvers to Dhalion using the appropriate APIs.

The Resolution Phase consists of two steps. In the first step, the diagnoses produced by the Diagnosis Generation Phase are examined in order to determine the appropriate Resolver that must be invoked. For example, as discussed before, the cause of backpressure could be attributed to various reasons, such as limited parallelism, slow instances or data skew. Depending on the diagnoses produced by various Diagnosers, this step explores the candidate Resolvers and selects the one that is more likely to solve the backpressure problem. Note that depending on the particular policy, the Resolver selection step can be performed using various methods such as rule-based or machine learning techniques. For example, given a set of diagnoses, one might always pick the Resolver that is most likely to solve the problem, or might decide to occasionally explore the space of Resolvers and pick an alternate one. When a user creates a new policy using the Dhalion APIs, she must also implement the policy's Resolver selection method that will be invoked in this step.

In the second step, the selected Resolver is invoked and performs the appropriate topology changes. Note that major topology changes such as scaling up and down resources or restarting containers, are typically invoked through the Heron `Scheduler` component and thus as shown in Figure 2, the `Resolvers` that perform such actions communicate with the Heron `Scheduler`.

Apart from the extensibility aspects, it is worth noting that a major advantage of Dhalion's policy architecture is its modularity. For example, instead of creating a monolithic `Diagnoser` that generates a diagnosis by evaluating all the symptoms, we instead decided to create multiple independent `Diagnosers`, each one evaluating a specific set of symptoms. This approach has two major advantages. First, it allows reusability of the `Diagnosers` by multiple policies as it is easier to combine different `Diagnosers` to address the needs of a particular policy. Second, it facilitates debugging and maintenance of the policy code since the users can easily debug and tune only specific `Diagnosers` without having to understand other parts of the source code. For the same reasons, the policy architecture and corresponding APIs incorporate multiple independent `Symptom Detectors` and `Resolvers`.

Another important aspect of Dhalion is that it is unaffected by noisy data and transient changes. This is mainly due to two reasons. First, the `Symptom Detectors` take into account multiple readings of a given metric over a large time period (e.g., 300 seconds) and thus are not typically affected by outliers. Second, after an action is performed by a Dhalion policy, the `Health Manager` waits for some time for the topology to stabilize before invoking again the policy. In this way, metric changes that occur while the action is taking place are not taken into account when the `Health Manager` re-evaluates the state of the topology and detects symptoms that can potentially denote that the topology is not in a healthy state. In Section 6, we experimentally show that by using these techniques, Dhalion remains unaffected by transient changes.

## 4.2 Action Log

The `Action Log` is a log that captures the actions taken by the `Health Manager` during policy execution. The log can be used for debugging and tuning a particular policy and for reporting statistics about the policy actions to the users or system administrators. As we see in the next section, the `Action Log` can also be useful when evaluating the effectiveness of a policy.

Each entry in the log contains the type of action that was taken by the policy. For example, if the policy invoked the `Scale Up Resolver`, a "scale up" action will be written to the log. The log entry also captures the time the action was taken as well as the diagnosis that led to this particular action. The users can manage the size of the log by configuring a log purge operation. More specifically, they can choose to keep the *n* most recent log entries or keep the log entries corresponding to the last *m* hours.

## 4.3 Action Blacklist

Dhalion maintains a blacklist of diagnosis descriptions and corresponding actions taken that did not produce the expected outcome. These actions will not be invoked again for a similar diagnosis during the execution of the policy. In particular, after a diagnosis has been produced and a corresponding action was taken by the policy, the `Health Manager` waits for some time to allow the topology to reach a steady state and then evaluates the action that was taken by obtaining the necessary information from the `Action Log`. When a new policy is defined, an evaluation method for this particular policy must be provided. For example, if a policy takes actions in an attempt to maximize throughput, the evaluation method of the policy can simply check whether an action has resulted in a throughput increase. This can be achieved by comparing the current topology state with the previous state captured in the diagnosis description of the `Action Log` entry that corresponds to the last action taken by the policy.

The system currently tracks the ratio of the number of times a particular action has not been beneficial for a given diagnosis over the total number of times the action has been invoked because of the diagnosis. When the ratio is higher than a configurable threshold, the diagnosis-action pair is placed in the `Action Blacklist`. During the `Resolution Phase` of the policy, before the selected `Resolver` is invoked, Dhalion automatically checks whether this action has been blacklisted for a similar diagnosis. If the action is already contained in the `Action Blacklist`, then the selected `Resolver` is not invoked. In such cases, the users can specify the behavior of the policy through the `Resolver` selection method that they define when creating the policy. For example, a user might decide to invoke another `Resolver` or wait until the policy is again executed by the `Health Manager`, possibly on a new topology state.

Finally, although Dhalion provides support for the `Action Blacklist`, users have the flexibility to decide whether they want to enable this mechanism when executing their policy.

## 5. DHALION USE CASES

As discussed in Section 4, Dhalion is a modular and extensible system that allows users to implement their own policies in order to meet their application requirements. In this section, we present two use cases of Dhalion and extensively discuss the implementation of the corresponding Dhalion policies on top of Heron. Note that our policies can also be applied to other streaming engines as long they employ a backpressure-based control rate mechanism.

## 5.1 Dynamic Resource Provisioning

Streaming jobs are typically long running with a time span of weeks or even months. During the application's lifecycle, the data load the system observes can change significantly over time. For example, the amount of data that needs to be processed in Twitter's data centers can vary significantly due to expected and unexpected global events. During these events, there are spikes of tweets that need to be processed in real-time. Users and system administrators typically overprovision the resources assigned to each topology so that the workload spikes can be efficiently handled. However, this approach is suboptimal as it can significantly increase the operating cost. Ideally, resources should be automatically scaled up and down to efficiently handle load variations while avoiding resource underutilization. For this reason, we created an *invasive* Dhalion policy, namely the `Dynamic Resource Provisioning Policy`, that observes the system behavior and dynamically provisions the topology resources so that the overall throughput is maximized while at the same time the resources are not underutilized.

The major goal of the `Dynamic Resource Provisioning Policy` is to scale up and down topology resources as needed while still keeping the topology in a steady state where backpressure is not observed. This is because the existence of backpressure denotes that the spouts are stalled which in turn means that the throughput is not maximized. The policy uses various `Symptom Detectors` and `Diagnosers` to determine whether the reason for topology instability is lack of resources or to explore opportunities for scaling down resources without sacrificing performance. Similarly, it employs various `Resolvers` that attempt to address the diagnosed problems. Figure 3 shows an overview of the policy's phases. We now discuss these phases in more detail.

**Symptom Detection Phase**: As shown in the figure, the policy employs three `Symptom Detectors` namely the `Pending Packets Detector`, the `Backpressure Detector` and the

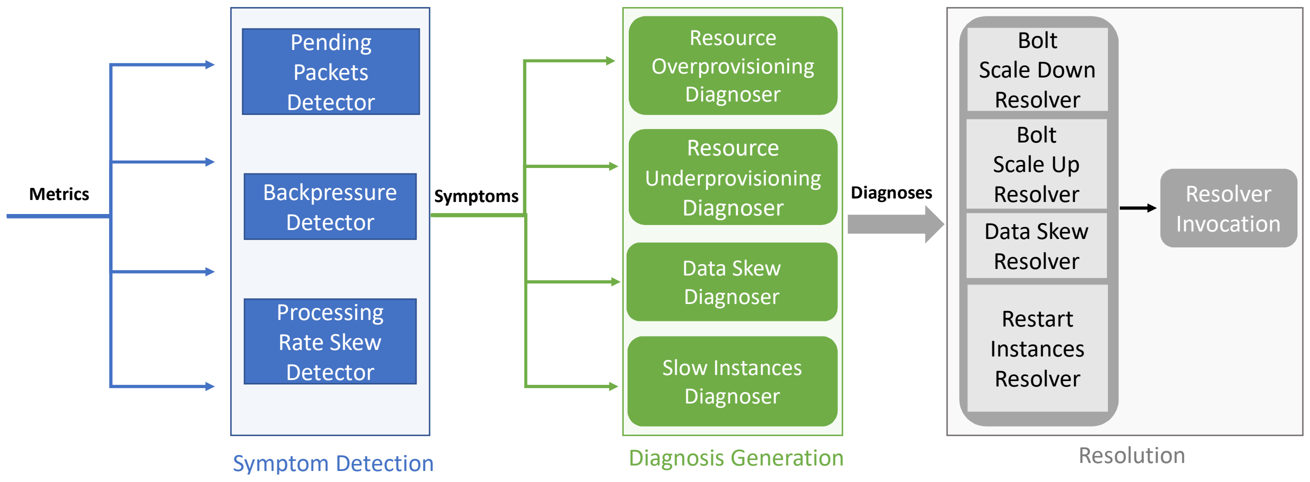

**Figure 3: Dynamic resource provisioning policy**

`Processing Rate Skew Detector`. Every time the policy is invoked by the Dhalion `Health Manager`, the `Symptom Detectors` evaluate various metrics over a 300 second interval and attempt to identify symptoms that can potentially denote that topology is not in a healthy state. The `Pending Packets Detector` focuses on the `Stream Manager` queue corresponding to each `Heron Instance`. Each `Stream Manager` queue temporarily stores packets that are pending for processing by the corresponding `Heron Instance`. This `Symptom Detector` examines the number of pending packets in the queues of the `Heron Instances` that belong to the same bolt, and denotes whether these `Heron Instances` have similar queue sizes or whether outliers are observed. As we discuss later, the queue sizes can provide insights about potential system bottlenecks. The `Backpressure Detector` examines whether the topology experiences backpressure by evaluating the appropriate `Stream Manager` metrics. If backpressure is observed, then the `Backpressure Detector` generates a symptom description that consists of the particular bolt that is the source of backpressure as well as the amount of time input data consumption was suspended during the 300 second measurement period. As discussed before, the existence of backpressure shows that the system is not able to achieve maximum throughput. Finally, the `Processing Rate Skew Detector` examines the number of tuples processed by each `Heron Instance` during the measurement period (processing rate). It then identifies whether skew in the processing rates is observed at each topology stage.

**Diagnosis Generation Phase**: The symptom descriptions produced by the `Symptom Detectors` are then forwarded to a set of `Diagnosers` as shown in Figure 3. The `Resource Overprovisioning Diagnoser` is typically useful when the input data load decreases and its major goal is to identify opportunities for scaling down resources when the topology is in a healthy state. The `Diagnoser` takes as input the symptom descriptions produced by the `Pending Packets Detector` and the `BackPressure Detector` and examines whether resources have been overprovisioned for the topology. More specifically, the `Resource Overprovisioning Diagnoser` first checks whether backpressure exists in the topology. In this case, it does not produce a diagnosis since the topology is not in a healthy state that would allow to identify opportunities for revoking resources. If backpressure is not observed, then the `Resource Overprovisioning Diagnoser` examines the average number of pending packets for the topology's `Heron Instances` based on the symptom description produced by the `Pending Packets Detector`. If the average number of pending packets for each `Heron Instance` of a bolt is almost zero then it is possible that the resources assigned to this bolt are overprovisioned and thus revoking some resources might not have a negative effect on the overall throughput. The `Resource Overprovisioning Diagnoser` examines all the bolts of the topology and produces a diagnosis description that describes which bolts might be overprovisioned. The description additionally contains information about the status of the remaining bolts with respect to the number of packets pending for their corresponding `Heron Instances`.

When the input load increases (workload spike), the topology might experience backpressure since the `Heron Instances` might become overloaded and thus, more resources must be provisioned to accommodate a larger number of `Heron Instances`. However, the existence of backpressure might not necessarily be attributed to insufficient resources. As mentioned in Section 3, slow instances or data skew in the workload might also cause backpressure. Thus, carefully examining the underlying cause of backpressure is crucial since this can help avoid unnecessary allocation of additional resources. The remaining three `Diagnosers` operate on topologies that experience backpressure and attempt to identify the cause of backpressure. As their names denote, the `Resource Underprovisioning Diagnoser` examines whether backpressure can be attributed to underprovisioned resources for the bolt that is the source of backpressure. The `Slow Instances Diagnoser` examines whether one or more `Heron Instances` of the bolt that initiated the backpressure are running slower than their peers. In this case, the slow `Heron Instances` might be the reason for the backpressure. Finally, the `Data Skew Diagnoser` examines whether one or more `Heron Instances` receive more data than their peers because of data skew and thus they are overloaded. Note that it is possible that the effects of data skew or slow instances are not significant enough to enable the backpressure mechanism of Heron and thus do not have a negative impact on the overall throughput. Our `Diagnosers` will not be able to diagnose such scenarios since they operate only over topologies that are not in a healthy state.

The three `Diagnosers` take into account the symptom descriptions produced by the `Backpressure Detector`, the `Pending Packets Detector` and the `Processing Rate Skew Detector` when generating a diagnosis. The `Diagnosers` first check whether

**Table 1: The logic of various Diagnosers**

| Diagnosis | Condition |
|---|---|
| Resource Underprovisioning | $\forall h_i, h_j \in \mathcal{H}$ : $r_i \simeq r_j$ and $p_i \simeq p_j$ |
| Slow Instances | $\forall h_i, h_j \in \mathcal{H} : r_i \simeq r_j$ and $\sum_{h_i \in \mathcal{B}} p_i/\|\mathcal{B}\| > \sum_{h_i \in \mathcal{H}-\mathcal{B}} p_i/\|\mathcal{H}-\mathcal{B}\|$ |
| Data Skew | $\sum_{h_i \in \mathcal{B}} r_i/\|\mathcal{B}\| > \sum_{h_i \in \mathcal{H}-\mathcal{B}} r_i/\|\mathcal{H}-\mathcal{B}\|$ and $\sum_{h_i \in \mathcal{B}} p_i/\|\mathcal{B}\| > \sum_{h_i \in \mathcal{H}-\mathcal{B}} p_i/\|\mathcal{H}-\mathcal{B}\|$ |

backpressure exists. If backpressure is not observed then they do not produce a diagnosis. Otherwise, they first examine which bolt initiated the backpressure. Then, they collect information about the processing rates of the `Heron Instances` of this bolt and their corresponding average number of pending packets. Let $\mathcal{H}$ be the set of `Heron Instances` corresponding to the bolt. Then, for each `Heron Instance` $h_i \in \mathcal{H}$, let $r_i$, $p_i$ be its corresponding processing rate and average number of pending packets, respectively. Also let $\mathcal{B}$ be the subset of `Heron Instances` that suspended data consumption during the measurement interval ($\mathcal{B} \subset \mathcal{H}$). Table 1 provides a description of the conditions that must be true so that the corresponding diagnosis is produced by the appropriate `Diagnoser`.

As shown in the table, when all the `Heron Instances` of the bolt have similar processing rates and their corresponding queues have similar sizes, then the `Resource Underprovisioning Diagnoser` determines that the observed symptoms are due to limited resources assigned to the bolt under consideration. This is because in cases where the bottleneck is the limited number of `Heron Instances` at a particular topology stage, all the `Heron Instances` of this stage would be overloaded and thus they would exhibit similar behaviors. The `Slow Instances Diagnoser` determines that the cause of backpressure is the existence of slow instances when the `Heron Instances` that initiated the backpressure have a much higher number of pending packets than their peers while operating at similar processing rates. Intuitively, one would expect the slow instances to have lower processing rates than their peers. However, because the slow instances initiate backpressure, the remaining instances operate at the speed of the slow instances without reaching their maximum capacity. Finally, the `Data Skew Diagnoser` attributes the existence of backpressure to data skew when the `Heron Instances` that initiated the backpressure have a higher processing rate and a higher number of pending packets than their peers. In case of data skew, some `Heron Instances` receive more data than their peers. If these instances do not have the processing capacity required to handle the input load, the number of their corresponding pending packets will be higher than that of their peers. Moreover, if their peers do not receive enough data to operate at full capacity, these `Heron Instances` will have higher processing rates than their peers since they process more data over the same time interval.

It is worth noting that the above techniques can correctly categorize backpressure-related problems as long as we can accurately detect outliers. The outlier detection methods typically categorize data based on some threshold. As a result, an incorrect diagnosis might be produced if the threshold is not set correctly. For example, when a `Heron Instance` is slightly slower than its peers, the outlier detection method might not be able to detect the problem. In this case, the `Slow Instances Diagnoser` will not produce a diagnosis but the `Resource Underprovisioning Diagnoser` will produce one. Our policy is able to address such scenarios by making use of the `Action Blacklist` and the appropriate `Resolver` selection method as we discuss later. Finally, we'd like to point out that only one out of the three conditions presented in Table 1 can be true at a time and as a result, only one `Diagnoser` will produce a diagnosis. As we discuss later, this observation simplifies the `Resolver` selection methodology.

**Resolution Phase**: In this phase, the diagnoses produced by the previous phase are examined in order to determine which `Resolver` to invoke. The policy employs four `Resolvers`. The `Bolt Scale Down Resolver` scales down resources of a particular bolt by decreasing the number of `Heron Instances` that correspond to the bolt. This `Resolver` is invoked when the `Resource Overprovisioning Diagnoser` produces a diagnosis. Note that if this `Diagnoser` generates a diagnosis then it is guaranteed that the remaining `Diagnosers` will not produce one. This is because the `Resource Overprovisioning Diagnoser` operates on a healthy topology whereas the remaining ones address backpressure-related problems. Automatically computing the scale down factor is challenging since we cannot predict the behavior of the topology as resources get revoked. Thus, in our current implementation the scale down factor is configurable. Hovever, note that if a scale down operation results in a state where backpressure is observed, the operation will be blacklisted and the policy will subsequently invoke a scale up operation to bring the topology back to a healthy state.

The `Restart Instances Resolver`, `Data Skew Resolver`, and `Bolt Scale Up Resolver` address the diagnoses produced by the `Slow Instances Diagnoser`, the `Data Skew Diagnoser` and the `Resource Underprovisioning Diagnoser` respectively. More specifically, the `Restart Instances Resolver` moves the slow `Heron Instances` to new containers whereas the `Data Skew Resolver` adjusts the hash function used to distribute the data to the bolts. We now explain in more detail the `Bolt Scale Up Resolver` since it is typically invoked when workload spikes are observed. This `Resolver` is responsible for scaling up the resources of the bolt that initiated the backpressure by automatically increasing the number of `Heron Instances` belonging to this bolt. To determine the scale up factor, the `Resolver` computes the percentage of the total amount of time that the `Heron Instances` spent suspending the input data over the amount of time where backpressure was not observed. This percentage essentially denotes the portion of the input load that the `Heron Instances` could not handle. For example, if 20% of the time the data consumption was suspended whereas 80% of the time the data flow was normal, then the `Heron Instances` were not able to handle $1/4$ of the input load and thus a 25% increase in parallelism is required. After determining the scale up factor, the `Resolver` invokes the appropriate Heron APIs to scale up the topology resources accordingly.

It is worth noting that since only one of the conditions presented in Table 1 is always true, only one `Diagnoser` produces a diagnosis each time backpressure is observed. Thus, this policy's `Resolver` selection method is straightforward as only one `Resolver` can address the backpressure problem. Note that if the selected `Resolver` is blacklisted for a particular diagnosis, then the `Resolver` selection method will randomly select one of the remaining two `Resolvers`.

## 5.2 Satisfying Throughput SLOs

We observe that in a large number of streaming applications, users spend a significant amount of time tuning the topology to meet the requirement of a throughput above a certain threshold. This is because either they are not aware of the number of spouts needed to fetch data at the required rate or they manually reconfigure the parallelism of the bolts to alleviate backpressure. In this section, we present the `Throughput SLO Policy` which addresses this problem. More specifically, the users who want to deploy a topology can use this policy to automatically configure the number of `Heron Instances` both at the spout and the bolt level so that their specific throughput SLO is satisfied.

The `Throughput SLO Policy` takes as input a throughput SLO that denotes the total rate at which the spouts should emit data. For example, a user might want to handle an input load of 3 million tuples/minute. The policy keeps tracking the actual throughput observed while the topology is running, and automatically adjusts the parallelism of spouts or bolts with the goal of satisfying the performance SLO. This policy can significantly reduce the time the users spend tuning the topology; users can simply submit a topology that contains spouts and bolts consisting of a single `Heron Instance` (parallelism = 1), and let the policy tune the parallelism of the various topology stages, so that the performance objective is met.

The `Dynamic Resource Provisioning Policy` that we previously presented, assumes that the input data rate is given and attempts to allocate resources so that the system can handle the input data load as it varies over time. The `Throughput SLO Policy` goes a step further by attempting to adjust the input data rate by increasing the number of `Heron Instances` that belong to the spouts, in order to meet the performance SLO. Note that it is possible that an SLO cannot be met because there is not enough data to satisfy the requirement. In this case, the `Throughput SLO Policy` will generate an alert. It is also possible that the policy might not be able to increase the parallelism of the spouts beyond a certain threshold. This typically happens when the spouts consume data from systems like Kafka [3]. In such cases, the parallelism of the spouts is limited by the number of the provisioned Kafka partitions. The policy will not attempt to increase the spout parallelism beyond this upper bound and will inform the user.

The `Throughput SLO Policy` reuses the components of the `Dynamic Resource Provisioning Policy` since it might scale up the resources assigned to the bolts of the topology. Apart from these components, the `Throughput SLO Policy` uses an additional `Symptom Detector`, `Diagnoser` and `Resolver`. More specifically, the `Emit Count Detector` computes the total rate at which spouts emit data and forwards it to the `Throughput SLO Violation Diagnoser`. The `Diagnoser` first checks whether the topology is in a healthy state. If backpressure is observed, the `Diagnoser` does not produce a diagnosis. Otherwise, it examines whether the current throughput meets the user's performance requirements. In case the user's SLO is violated, the `Diagnoser` produces a diagnosis description that is forwarded to the `Spout Scale Up Resolver` which in turn increases the number of `Heron Instances` of the spout. To determine the scale up factor, the `Resolver` divides the user's throughput requirement by the currently observed throughput.

In case the policy increases the spout parallelism, the topology might experience backpressure due to the increase of the input load. In this case, the `Throughput SLO Policy` employs the components used by the `Dynamic Resource Provisioning Policy` to automatically adjust the resources assigned to the bolts so that the topology is brought back to a healthy state. In Section 6, we experimentally evaluate the `Throughput SLO Policy`.

## 6. EXPERIMENTAL EVALUATION

In this section, we evaluate our Dhalion policies and provide an analysis of the experimental results. We first evaluate the `Dynamic Resource Provisioning Policy` and then the `Throughput SLO Policy`. Our key results are the following:

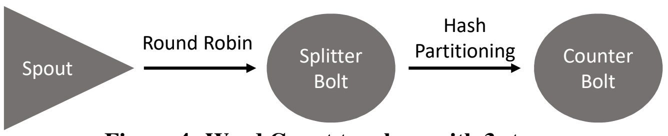

**Figure 4: Word Count topology with 3 stages**

1. Dhalion works well for multi-stage topologies where backpressure propagates from one stage to the other (see Figures 5, 7, 9, 10).
2. The system is able to dynamically adjust resources when load variations occur while still reaching a steady state where throughput is maximized (see Sections 6.2).
3. The system is able to automatically reconfigure a topology in order to meet a user-specified throughput SLO, even in cases where the user did not spend any time tuning the topology (see Section 6.3)
4. Dhalion's actions are unaffected by noise and transient changes (see Section 6.2).
5. Dhalion can bring the topology to a healthy state even when multiple problems occur (see Section 6.5).

In the following sections, we describe our experimental setup and further analyze our findings.

## 6.1 Experimental Setup

**Hardware and Software Configuration:** All our experiments were performed on Microsoft HDInsight [8] on top of Azure Instances of type D4. Each D4 instance has one 8-core Intel Xeon E5-2673 CPU@2.40GHz and 28GB of RAM and runs Ubuntu version 16.04.2. In all our experiments we use Heron version 0.14.5 on top of YARN version 2.7.3.

**Heron Topology:** Previous work on streaming systems [22, 26] used a 2-stage `Word Count` topology to evaluate the systems under consideration. In our work, we decided to use a 3-stage `Word Count` topology that operates at the level of sentences and not single words as the corresponding 2-stage topology. In this way, we can demonstrate that our Dhalion policies can handle topologies where backpressure propagates from one stage to another. In our topology, the spout generates a 200 character long sentence by randomly selecting words from a set of 450K English words and emits it. The spouts distribute the sentences to the bolts belonging to the $2^{nd}$ stage of the topology (`Splitter` bolts) in a round robin fashion. The `Splitter` bolts split the sentences into words that subsequently forward to the $3^{rd}$ stage bolts (`Counter` bolts) using hash partitioning. Finally, the `Counter` bolts count the number of times each word was encountered.

**Evaluation Metrics:** In our experiments, we often use throughput as an evaluation metric. We note that the throughput for a spout is defined as the number of tuples *emitted* by the spout over a period of one minute. The throughput of a bolt is defined as the number of tuples *processed* by the bolt over an one minute period. To track resource allocation over time, we present the number of `Heron Instances` provisioned at each topology stage.

## 6.2 Dynamic Resource Provisioning

In this experiment, we analyze the behavior of the `Dynamic Resource Provisioning Policy`. We start by deploying the

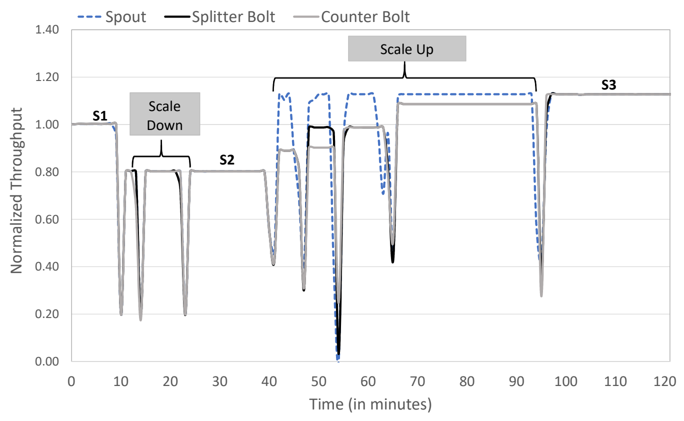

**Figure 5: Dhalion's reactions during load variations**

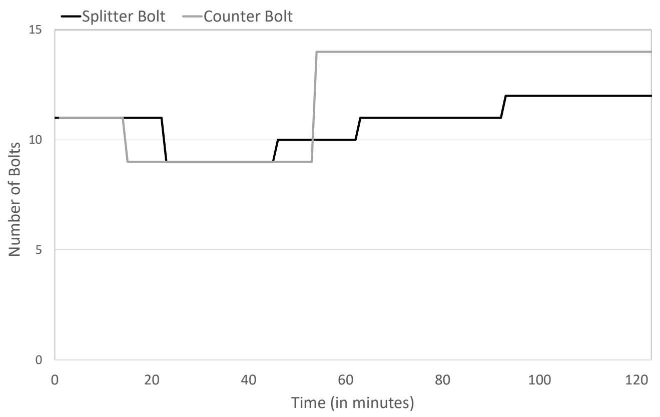

**Figure 6: Number of Heron Instances provisioned during load variations**

Word Count topology using 40 spouts, 11 Splitter bolts and 11 Counter bolts. In this state the topology does not experience backpressure. We refer to this initial state as **S1**. We then decrease the input load by 20% by manually reducing the number of spouts and observe whether the policy invokes the appropriate scale down operations. After some time, we increase the input load by 30% and observe again the behavior of the policy. Note that we intentionally avoid introducing dramatic load variations in order to demonstrate that our Dhalion policy is capable of identifying smaller load variations and adjust resources accordingly. The policy is invoked every 2 minutes and monitors the topology state. When the policy performs an action, the Health Manager waits for a few minutes for the topology to stabilize before invoking the policy again.

Figure 5 shows the normalized throughput at each topology stage during the execution of the experiment; the numbers plotted are normalized to the corresponding throughput observed when the topology was at state **S1**. Figure 6 shows the corresponding number of Heron Instances belonging to the Splitter and Counter bolts during the execution of the experiment.

For the first 10 minutes the topology is at stable state **S1**. Then, we manually decrease the input load by setting the number of spouts to 32. At this point, the throughput temporarily becomes zero since Heron stops processing new data while a parallelism change happens. After the change is completed, the throughput comes back to normal level. Note that the throughput observed after the parallelism decrease is lower than that of state **S1** since the spouts emit fewer tuples. At this point, the policy successfully detects that there is opportunity for scaling down resources. In particular, it first detects that the number of pending packets for the Heron Instances of the Counter bolt are almost zero and thus, it invokes the Scale Down Resolver at minute 14. As shown in Figure 6, the Resolver removes two Heron Instances of the Counter bolt bringing the number of instances down to 9. Note that after this change, the observed throughput is the same as before. This clearly demonstrates that the policy was successful since it reduced the overall resources without sacrificing performance. Also note that Dhalion correctly decided to scale down resources despite the fact that a few minutes ago there was a dramatic decrease in throughput. *This demonstrates that Dhalion is unaffected by transient changes or noisy data.*

After the scale down operation is performed, the Health Manager waits for sometime before invoking the policy again. At minute 23, the policy is invoked and detects that there is opportunity for scaling down the resources assigned to the Splitter bolt. It then removes two Heron Instances bringing the number of instances down to 9. As seen in the figure, the throughput still remains at the same levels indicating that the policy correctly adjusted the parallelism. After this change, the policy does not detect other opportunities for scaling down and thus the topology operates at stable state **S2**. At this state, both the Splitter and the Counter bolt consist of 9 Heron Instances each.

At minute 40, we manually increase the number of spouts to 45, thus increasing the data load. As shown in Figure 5, after the parallelism change, there is a gap between the throughput of the spout and that of the bolts. This is because the Stream Manager queues that hold the packets that are pending for processing at the Splitter bolt keep accumulating packets. The queue sizes keep increasing for about 5 minutes until a threshold is reached and backpressure is invoked. At minute 46, the policy detects the backpressure, determines that the Splitter bolt is the bottleneck and increases its parallelism by 1. After that change, the topology does not experience backpressure and the throughput of the Splitter bolt increases. Now the Counter bolt experiences higher load and its corresponding queues start accumulating more packets. At minute 53, the Counter bolt initiates backpressure. The policy detects the backpressure and attributes it to the limited number of Heron Instances at the last stage of the topology. As a result, it scales up the resources assigned to the Counter bolt by increasing its parallelism from 9 to 14. The topology requires two more rounds of scaling before achieving stable state (**S3**). More specifically, the Splitter bolt initiates backpressure at minute 65 and minute 93. The policy correctly increases the bolt's parallelism by 1 in both cases increasing the total number of Heron Instances provisioned for this bolt to 12. At state **S3**, the Splitter and Counter bolt consist of 12 and 14 Heron Instances respectively.

As we mention in Section 5, the policy scales up resources only when backpressure is observed. As shown in this experiment, backpressure will not be initiated unless the size of at least one Stream Manager queue increases beyond a threshold. However, the process of filling up the queues might take some time during which our policy will not initiate any scale up operation. This is typically acceptable in the context of long-running streaming applications, especially during the initial configuration phase where topologies are extensively tuned before being deployed in production for weeks or even months. However, as part of future work, we plan to further improve our algorithms by taking into account the rate at which the

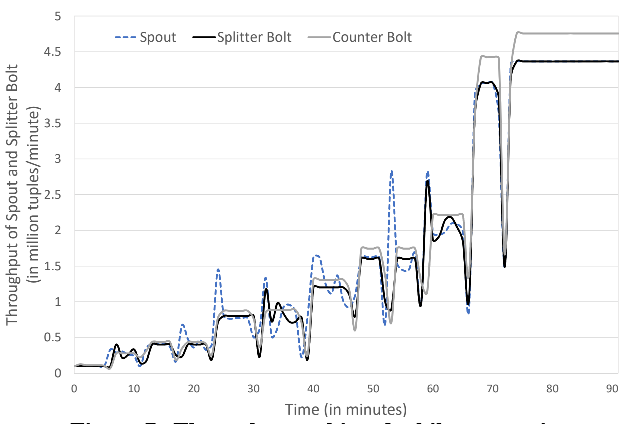

**Figure 7: Throughput achieved while attempting to satisfy a throughput SLO**

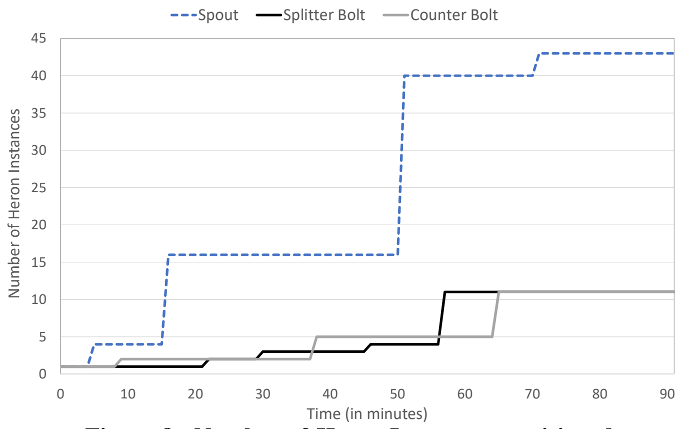

**Figure 8: Number of Heron Instances provisioned while attempting to satisfy a throughput SLO**

average queue sizes change. In this way, the policy's reaction time can potentially be further minimized.

Our experiment shows that the `Dynamic Resource Provisioning Policy` is able to adjust the topology resources on-the-fly when workload spikes occur. Moreover, the policy is able to eventually reach a healthy state where backpressure is not observed and the overall throughput is maximized. Finally, the policy can correctly detect and resolve bottlenecks even on multi-stage topologies where backpressure is gradually propagated from one stage of the topology to another.

## 6.3 Satisfying Throughput SLOs

In this experiment, we use the `Word Count` topology to evaluate our `Throughput SLO Policy`. We evaluate a scenario where the user does not tune the parallelism of the topology before deploying it, but instead provides a throughput SLO and expects the policy to automatically configure the topology. The topology is initially submitted with a single `Heron Instance` provisioned for the spout and each of the bolts (parallelism =1). As part of the policy's configuration, the user specifies that the topology should handle at least 4 million tuples/minute at a steady state.

Figure 7 shows how the throughput of the spouts and the bolts adjusts over time. The SLO is defined based on the number of tuples that are emitted by the spouts and thus once the blue line reaches the desired level and the system is at a steady state, the `Throughput SLO Policy` will not make any further adjustments. Note that the throughput of the `Counter` bolt is much higher than that of the `Splitter` bolt since the latter operates on top of sentences whereas the former on top of the words contained in those sentences and thus, it receives a much higher number of tuples. For this reason, the throughput of the `Counter` bolt is plotted separately on the right y-axis. Figure 8 shows the corresponding number of `Heron Instances` at each topology stage during the experiment.

As shown in Figure 7, the policy applies several actions until the throughput of the spout reaches the desired level. More specifically, the policy increased the number of spouts 4 times. It also increased the number of `Splitter` and `Counter` bolts, 4 times and 3 times, respectively. In the beginning of the experiment, the policy observes that the actual throughput is much less than the desired one and thus it decides to increase the number of spouts to 4. Note that we have configured the policy to scale up the number of spouts at most by a factor of 4 at a time in order to gradually propagate the load change to the later stages of the topology. After we increased the number of spouts, the system experienced backpressure initiated by the `Counter` bolt and thus the policy assigned an additional `Heron Instance` to this bolt. At this time, the topology is in a healthy state but the desired throughput is not yet reached. The policy detects the problem and at minute 16, increases again the number of spouts to 16. After this parallelism change, backpressure is propagated between the `Splitter` and `Counter` bolts causing four scale up operations from minute 22 until minute 46. Figure 8 shows the number of `Heron Instances` for each bolt during this time interval. From minute 51 until minute 71, the policy invokes two more spout scale up operations and handles the existence of backpressure by scaling up the bolts that initiated it. The topology reaches steady state when the throughput observed is equal or higher to the throughput SLO and backpressure is not observed. During steady state, the observed throughput is about 4.3 milion tuples/minute and the topology consists of 43 spouts, 11 `Splitter` bolts and 11 `Counter` bolts.

Our experiment shows the `Throughput SLO Policy` can successfully auto-tune the topology so that the throughput SLO is met.

## 6.4 Evaluating the Diagnosers

The previous experiments demonstrated the effectiveness of the `Resource Overprovisioning Diagnoser` and the `Resource Underprovisioning Diagnoser` in detecting opportunities for scaling up and down resources. In this experiment, we evaluate the effectiveness of the `Slow Instances Diagnoser` and `Data Skew Diagnoser`. More specifically, we synthetically generate `Word Count` topologies that either exhibit data skew or contain a `Heron Instance` that is slower than its peers. We then evaluate whether the `Diagnosers` of Dhalion's `Dynamic Resource Provisioning Policy` are able to correctly diagnose the cause of backpressure.

To generate topologies with a slow `Heron Instance`, we altered the average tuple processing latency of one instance of the `Splitter` bolt to make it perform slower than its peers. This slowdown was achieved by introducing appropriate sleep operations. As a result, an $X\%$ slower instance has a peak processing rate that is $X\%$ lower than that of its peers. To generate topologies with data skew, all the spout instances were configured to increase the frequency of a specific word in the emitted sentences. To synthetically create a topology with $X\%$ data skew, a word appears $X$ times in a collection of 100 words used to form sentences.

Table 2 presents our results. For each scenario examined, the second column of the table presents the ratio of the average process-

**Table 2: Effectiveness of the Diagnosers in various scenarios**

| Scenario | Processing Rate Ratio | Backpressure Percentage | Correct Diagnosis |
|---|---|---|---|
| 25% Slower Instance | 1 | 15% | Yes |
| 50% Slower Instance | 1 | 65% | Yes |
| 75% Slower Instance | 1 | 87% | Yes |
| 5% Data Skew | 1.4 | 24% | No |
| 15% Data Skew | 2.5 | 34% | Yes |
| 25% Data Skew | 4.1 | 90% | Yes |

ing rate of the `Heron Instances` that initiated backpressure over the average processing rate of the remaining `Heron Instances`. As discussed in Section 5.1, the `Diagnosers` produce a *slow instance diagnosis* when this ratio is almost 1. If the ratio is greater than 1 then a *data skew diagnosis* is produced. The table also contains information about the percentage of time that was spent suspending input data due to backpressure.

As shown in the table, the `Slow Instances Diagnoser` produced a successful diagnosis for the topologies with a slow instance, even when the instance was only 25% slower than its peers. Note that the ratios of processing rates observed in these scenarios are 1 as expected. Another interesting observation is that the backpressure percentage increases as the instance becomes slower. This behavior is expected since the slower the `Heron Instance` is, the more time it will suspend input data consumption.

The `Data Skew Diagnoser` produced a succesful diagnosis in all but one scenarios. As shown in the table, the ratio of the processing rates observed is greater than 1 as expected. However, when the data skew is small (5%), the `Diagnoser` did not consider the variance in the processing rates significant enough to justify a *data skew diagnosis* and thus a *slow instance diagnosis* was produced by the `Slow Instances Diagnoser`. However, since Dhalion employs the blacklist mechanism, the correct diagnosis was eventually produced even for this scenario.

## 6.5 Mixed Scenarios

In this experiment, we evaluate the `Dynamic Resource Provisioning Policy` in a mixed scenario where multiple problems occur. More specifically, the `Splitter` and `Counter` bolts are both underprovisioned. Additionally, the `Splitter` bolt contains a slow instance. We experimented with both 25% and 75% slower instances. In both experiments, the topology consists of 40 spouts, 8 `Splitter` and 8 `Counter` bolts when initially deployed and backpressure is observed.

Figures 9 and 10 show our results. As shown in Figure 9, at minute 8 the `Dynamic Resource Provisioning Policy` detects the slow `Splitter` bolt and restarts it. This action results in an increase of the total throughput. Hovever, because the `Splitter` bolt is also underprovisioned, backpressure is still observed. At minute 16, the policy scales up the resources of the `Splitter` bolt by increasing its parallelism from 8 to 11. After the scale up operation, the backpressure propagates to the `Counter` bolt which is also underprovisioned. Thus, at minute 24, the policy increases the number of `Counter` bolts to 12 which brings the topology to a healthy state where backpressure is not observed.

Figure 10 shows the behavior of the policy when there is a 75% slower instance. This scenario is more challenging since this instance is not significantly slower than its peers. As a result the policy, is not able to make a *slow instance diagnosis* but produces a *resource underprovisioning* diagnosis for the `Splitter` bolt instead. At minute 11, the policy invokes a scale up operation which brings the number of `Splitter` bolts to 15. At this point, the slow instance is not a bottleneck any more. However, backpressure is observed because the `Counter` bolt is underprovisioned. At minute 19, a scale up operation increases the number of `Counter` bolts to 11 which brings the topology to a healthy state. Note that ideally, the topology should end up having a similar configuration as in the 25% slower instance case where the slow instance problem was detected and resolved before the first scale up operation. The problem appears because the `Slow Instances Diagnoser` uses a threshold-based function to determine whether the `Splitter` bolts have similar behavior or whether outliers exist. In this experiment, the slow instance is not significantly slower than its peers. Thus, this function does not detect any outliers. As a result a *resource underprovisioning diagnosis* is produced which triggers a `Splitter` scale up operation. Note that this operation was not blacklisted since it resolved the backpressure problem. As part of future work, we plan to investigate whether machine learning techniques can produce more accurate diagnoses in such scenarios.

## 7. RELATED WORK

Initial work on stream data processing began about a decade ago when streaming engines such as STREAM [20], Aurora [13] and Borealis [11] were developed. Over the last few years the need for scalable streaming engines became more prominent as many business operations depend on real-time analytics. Several systems have been created [2, 4, 5, 9, 10, 12, 22, 26] and many of them, such as Heron [22], Storm [26], Samza [4], Spark Streaming [10] and Flink [2] have been open-sourced. These systems operate at large-scale and can tolerate various hardware and software failures.

However, despite the significant progress made over the years, none of the existing streaming systems are truly self-regulating. Dhalion addresses this issue by operating on top of the streaming engine and providing self-regulating capabilities to it. Note that although Dhalion has been implemented on top of Heron, its architecture and basic policy abstractions can be adopted by other streaming engines as long as they provide a metrics collection API and potentially a scaling API.

Dhalion provides the necessary abstractions to address performance variability problems due to performance variance in multiple hardware levels such as CPU and network I/O or software delivering degraded quality of service [15, 16, 25]. The policies presented in this paper automatically adjust the topology configuration so that the performance objectives are met even in the presence of slow machines/containers.

Similar to our dynamic resource provisioning policy, auto-scaling techniques for streaming applications have previously been proposed [18, 19]. These methods however are not directly applicable to systems that employ backpressure mechanisms to perform rate control. In such settings, one has to examine whether existence of backpressure can be attributed to resource underprovisioning or other factors. To the best of our knowledge, none of the existing open-source streaming systems performs automatic scaling based on the input load. The work in [24] presents an adaptive load manager that performs load shedding based on the observed response times. The Dhalion policies can potentially incorporate such load shedding techniques in order to avoid system overload.

Self-tuning techniques for databases and Map Reduce systems have been extensively studied in the past [14, 21]. Recent work proposes self-driving relational database systems that predict future workloads and proactively adjust the database physical design [23].

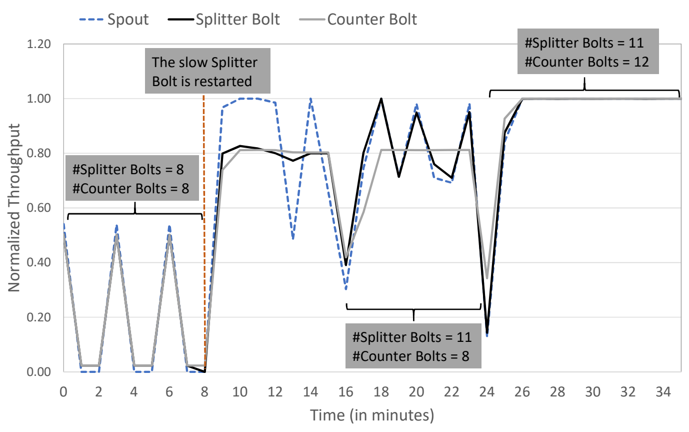

**Figure 9: Mixed scenario with underprovisioned resources and a 25% slower instance**

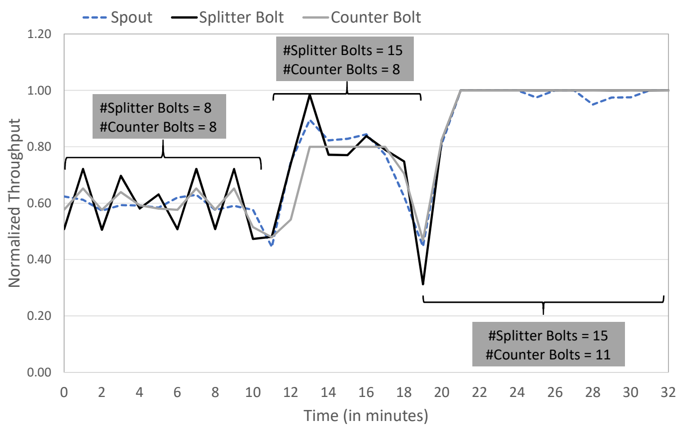

**Figure 10: Mixed scenario with underprovisioned resources and a 75% slower instance**

These works mainly focus on the self-tuning aspects of the systems and do not discuss mechanisms for creating self-stabilizing and self-healing systems. Moreover, the techniques presented in these studies are not directly applicable to streaming systems since they target different application scenarios.

## 8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce the notion of self-regulating streaming systems. We then present Dhalion, a modular and extensible system that is deployed on top of streaming systems and provides them with self-regulating capabilities through the execution of various Dhalion policies. Dhalion provides the necessary abstractions for users to implement their own policies and incorporate them in their streaming applications. We present a Dhalion policy that automatically scales up and down resources based on the input data load and a policy that auto-tunes the topology by provisioning the necessary resources to meet a throughput SLO. Both policies have been implemented and evaluated on top of Heron.

As part of future work, we plan to expand Dhalion's capabilities by incorporating more policies that satisfy various application requirements such as policies that enforce latency SLOs. We also plan to investigate whether Dhalion's decision making process can be further automated using machine learning models. Finally, an interesting direction for future work is to evaluate the applicability of Dhalion to other categories of Big Data engines, such as batch processing and machine learning systems.

## 9. REFERENCES

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

[2] Apache Flink. `https://flink.apache.org/`.

[3] Apache Kafka. `http://kafka.apache.org/`.

[4] Apache Samza. `http://samza.apache.org/`.

[5] DataTorrent. `https://www.datatorrent.com`.

[6] Distributed Log. `http://distributedlog.incubator.apache.org/`.

[7] Heron Code Repository. `https://github.com/twitter/heron`.

[8] Microsoft HDInsight. `https://azure.microsoft.com/en-us/services/hdinsight/`.

[9] S4 Distributed Data Platform. `http://incubator.apache.org/s4/`.

[10] Spark Streaming. `http://spark.apache.org/streaming/`.

[11] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[12] T. Akidau et al. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, Aug. 2013.

[13] H. Balakrishnan et al. Retrospective on Aurora. *The VLDB Journal*, 13:2004, 2004.

[14] S. Chaudhuri and V. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, September 2007.

[15] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[16] T. Do et al. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *SOCC '13*, pages 14:1–14:14. ACM, 2013.

[17] M. Fu et al. Twitter Heron: Towards Extensible Streaming Engines. In *ICDE*. IEEE, 2017.

[18] T. Z. J. Fu et al. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *ICDCS*, pages 411–420, 2015.

[19] B. Gedik et al. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1447–1463, June 2014.

[20] T. S. Group. STREAM: The Stanford Stream Data Manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[21] H. Herodotou et al. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.

[22] S. a. Kulkarni. Twitter Heron: Stream Processing at Scale. In *ACM SIGMOD '15*, pages 239–250, 2015.

[23] A. Pavlo et al. Self-Driving Database Management Systems. In *CIDR*, 2017.

[24] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Avoiding Class Warfare: Managing Continuous Queries with Differentiated Classes of Service. *The VLDB Journal*, 25(2):197–221, Apr. 2016.

[25] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1-2):460–471, Sept. 2010.

[26] A. Toshniwal et al. Storm@Twitter. In *2014 ACM SIGMOD*.

[27] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, pages 5:1–5:16. ACM, 2013.