# Dhalion in Action: Automatic Management of Streaming Applications

Ashvin Agrawal
Microsoft
USA
ashvin.agrawal@microsoft.com

Avrilia Floratou
Microsoft
USA
avflor@microsoft.com

## ABSTRACT

In a world where organizations are being inundated with data from various sources, analyzing data and gaining actionable insights in real-time has become a key service differentiator. Over the last few years, several stream processing frameworks have been developed to address the need for large-scale, real-time analytics. A crucial challenge in these environments is the complexity of configuring, managing and deploying long-running streaming applications. Operators must carefully tune these systems to balance competing objectives such as resource utilization and performance. At the same time, they must also account for external shocks such as unexpected load variations and service degradations.

In this demonstration, we show how operators can maintain healthy streaming applications without manual intervention while still meeting their performance objectives. We use Dhalion, an open-source library that sits on top of the streaming application, observes its behavior and automatically takes actions to keep the application in a healthy state. In particular, through various Dhalion policies that are configured by the attendees, we demonstrate how a streaming application can meet its performance objective by automatically configuring the amount of resources needed at the various application stages. We also demonstrate Dhalion's modularity and extensibility that greatly simplifies the process of developing new policies which address different application requirements.

**PVLDB Reference Format:**
Ashvin Agrawal and Avrilia Floratou. Dhalion in Action: Automatic Management of Streaming Applications. *PVLDB*, 11 (12): 2050-2053, 2018.
DOI: https://doi.org/10.14778/3229863.3236257

## 1. INTRODUCTION

In recent years, there has been an explosion of real-time analytics needs and a plethora of streaming systems have been developed to support such applications (Apache Storm [10], Spark Streaming [12], Twitter's Heron [9], etc). These systems can be deployed at large scale and can recover from hardware and software failures. Despite the advances in the design of these systems, a crucial challenge for the operators remain the complexity of configuring and managing the streaming applications. This process is typically manual and time consuming as it often requires balancing conflicting objectives such as various performance requirements and resource utilization while also considering external shocks such as unexpected load variations.

In our prior work [7], we presented Dhalion, an open-source library [3] that sits on top of the streaming application and provides self-regulating capabilities to it. Dhalion closely monitors the streaming application, collects various metrics and identifies problems that might affect the health of the application. For example, by correlating various application metrics, Dhalion identifies slow processes, data skew or under-provisioned components and automatically takes corrective action. Dhalion is flexible enough to incorporate various policies depending on the user requirements and objectives.

In this work, we use Dhalion to demonstrate how operators can automatically manage their streaming applications while meeting their performance goals. We employ various Dhalion policies which target different application scenarios and user requirements to highlight Dhalion's adaptive behavior. The attendees will be able to configure the Dhalion policies and observe the impact of their choices on the performance and resource utilization of the underlying streaming application. They will also be able to observe which parameters affect Dhalion's reaction time. Our demonstration is based on top of the Heron streaming engine [9]. Heron is an open-source, highly scalable streaming engine that has been developed and deployed in Twitter.

In this work, we make the following contributions:

1. Motivated by real challenges that users face when deploying streaming applications, we demonstrate Dhalion, a library that monitors the streaming application and automatically takes corrective action to resolve potential issues.
2. We show how operators and solution architects can use Dhalion to automatically tune their streaming applications to meet a specific throughput requirement. Through an interactive interface, we allow the attendees to specify their own performance objectives and observe how Dhalion reacts to their input.
3. We demonstrate how operators can maintain their application in a healthy state even in the presence of input

*Proceedings of the VLDB Endowment,* Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236257

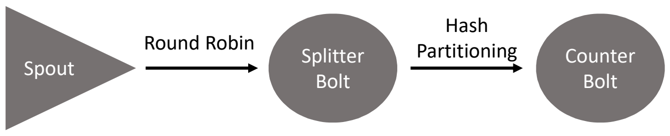

**Figure 1: Heron topology with three stages**

load variations. In particular, we develop three different auto-scaling policies that range from conservative to aggressive. The policies detect performance bottlenecks and dynamically provision the resources of various stages of the application. The three policies can result in different reaction times and resource utilization. The attendees will be able to vary the input load and observe the impact of their choices on the three policies as well as the application's performance.

In Sections 2 and 3, we provide a brief overview of Heron and Dhalion respectively. We then describe in detail our demonstration setup and scenarios in Section 4.

## 2. HERON BACKGROUND

Before describing Dhalion, we present a brief overview of Heron and its rate control mechanisms. This information is useful to better understand our demonstration scenarios. An extensive description of Heron can be found in [8, 9].

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 [2] or Distributed Log [4] 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 1 shows an example word count topology with three stages. The spouts distribute incoming sentences to the splitter bolts in a round robin fashion. The splitter bolts split the sentences into words that are subsequently forwarded to the counter bolts using hash partitioning. Finally, the counter bolts count the number of times each word was encountered.

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.

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. 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.

## 3. DHALION OVERVIEW

In this section, we provide a brief overview of Dhalion's main components and discuss its integration with the Heron streaming engine. The interested reader can find more information about Dhalion and its architecture in [7].

Dhalion's main goal is to keep the streaming application in a healthy state by detecting problems within the streaming application and automatically take corrective action to resolve the issues. Dhalion's architecture is inspired by common medical procedures. In particular, Dhalion first identifies *symptoms* that can potentially denote that a problem exists in the streaming application. After collecting all the symptoms, Dhalion attempts to find the root cause of the symptoms and produce a *diagnosis*. Finally, based on the diagnosis, Dhalion can automatically take the appropriate action to resolve the issues.

The above steps are executed through a Dhalion policy that is continuously executed while the streaming application is running. The phases of the Dhalion policy are described in Figure 2. As shown in the figure, Dhalion executes policies that consist of three parts. First, Dhalion collects metrics from the underlying streaming application and tries to identify symptoms that denote that the application might not be in a healthy state. The symptom detection is performed by various *detectors*. Each detector is a self-contained module that implements specific APIs and generates a specific set of symptoms. For example, in the context of Heron, the *backpressure detector* generates a symptom that denotes which instance in the topology generates backpressure. After the symptoms are generated, they are processed by the *diagnosers*. Each diagnoser is responsible for finding correlations among symptoms and evaluate if a specific diagnosis is valid. For example, the *under-provisioning diagnoser* evaluates whether the cause of the observed symptoms is resource under-provisioning at a particular stage in the topology, or the *slow instance diagnoser* evaluates whether one or more instances are slower than their peers in the same stage. At the last phase, a *resolver* is executed which is responsible for taking an action to bring the topology back to a healthy state. For example, if the topology is under-provisioned, the *scale up resolver* can automatically add more resources to the stage that is the bottleneck.

Note, that the *detectors*, *diagnosers* and *resolvers* implement well-specified Dhalion APIs which makes it easier to compose new policies. Additionally, since they are self-contained modules, they can be combined and re-used in different Dhalion policies which greatly simplifies the task of generating new policies. As we discuss in the following section, we will demonstrate various Dhalion policies that re-use some of these modules.

## 4. DEMONSTRATION OVERVIEW

The goal of the demonstration is to help the attendees understand how Dhalion can be used to maintain a healthy streaming application. Additionally, we intend to demonstrate Dhalion's flexibility in incorporating new policies as well as its modularity and extensibility. To this end, we deploy Dhalion on top of Heron and apply various Dhalion policies whose behavior is tailored to specific user needs such as meeting an service-level objective (SLO) or keeping the application in a steady state while input load variations oc-

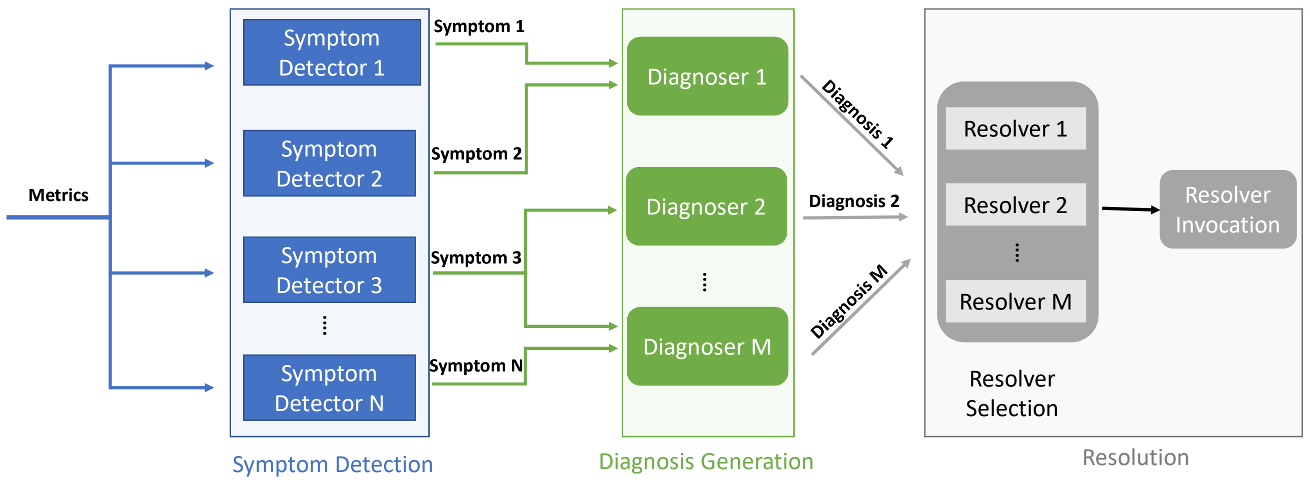

**Figure 2: Dhalion Policy**

cur. Additionally, we demonstrate the Dhalion abstractions and highlight its modularity.

## 4.1 Setup Details

**Hardware and Software Configuration.** We deploy Dhalion on top of Apache Heron (incubating) version 0.17.4 [1]. We use Hadoop Yarn [11] version 2.7.3 on Microsoft HDInsight [6] on top of Azure Instances of type D4. Each D4 instance has one 8-core Intel Xeon E5-2673CPU@2.40GHz and 28GB of RAM and runs Ubuntu version 16.04.2.

**Visualization Tools.** To visualize the impact of Dhalion policies on the underlying Heron application, we employ the Heron UI and Graphite [5]. Figures 3 and 4 show the two user interfaces. The Heron UI shows information about the topology graph, the topology deployment and some important system metrics. Graphite shows the variation of various topology metrics as the Dhalion policies adjust the topology to bring it back to a healthy and stable state. Note that the graphs produced by Graphite are continuously updated as Dhalion operates on the topology and thus attendees can gradually observe the behavior of the application.

**Workload.** Our workload consists of a topology that finds the top trending twitter hash tags. The topology reads incoming tweets, extracts the hash tags, and counts the number of times each hash tag has been used in a sliding time window. The hash tags are ranked based on their frequency and the top-k hash tags are emitted. This topology consists of four stages.

## 4.2 Demonstration Scenarios

Our demonstration scenarios place the attendee in the shoes of a solution architect/operator who is responsible for deploying streaming applications with specific performance requirements and for monitoring the application while it is active. This process is typically manual and involves extensive tuning before deploying the application in production, and close monitoring and manual intervention when problems arise while the application is online. Through this demonstration, the attendees will configure Dhalion to automatically keep their applications in a healthy state without manual intervention and they will be able to observe the impact of their choices on the streaming application using our visualization tools.

### *4.2.1 Tuning for meeting SLOs*

Often solution architects find themselves in a precarious ask; to configure and deploy a topology that meets a required throughput SLO without having a detailed performance profile of the various topology components. Tuning the topology for a particular SLO is a tedious and time-consuming task as the user needs to manually investigate the performance profile of the application through a trial and error procedure. Our discussion with users of streaming frameworks (including Heron) suggest that this manual tuning process can even take hours.

During the demonstration, the attendees will be able to observe the performance of a Dhalion policy that takes as input a throughput SLO requirement and automatically tunes the topology gradually until the SLO is met. In particular, the attendees will be able to configure the throughput SLO of the topology as well as the initial resources allocated to it. Through our visualization tools, the attendees will observe Dhalion gradually configuring the number of Heron Instances and their associated resources at each stage of the topology until the required SLO is met. By varying the initial resources allocated to the topology and the SLO value, the attendees can also observe the time needed for Dhalion to reconfigure the topology to meet the SLO.

### *4.2.2 Maintaining a healthy topology*

The previous demonstration scenario focused on automating the tuning process for the solution architect. In this scenario, we focus on maintaining the health of an application after it is tuned and deployed. Since streaming applications are typically long-running applications that might be active for weeks or even months, they are often faced with various external shocks that can threaten their stability. For example, it is common for Twitters tweet processing applications to experience loads that are several times higher than normal when users react to natural disasters. Since these load variations are typically unpredictable, operators are forced to either over-provision resources for these applications to

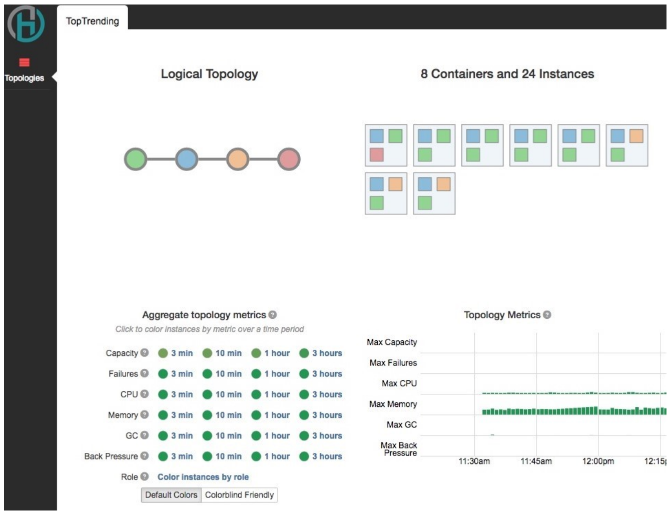

**Figure 3: Heron user interface**

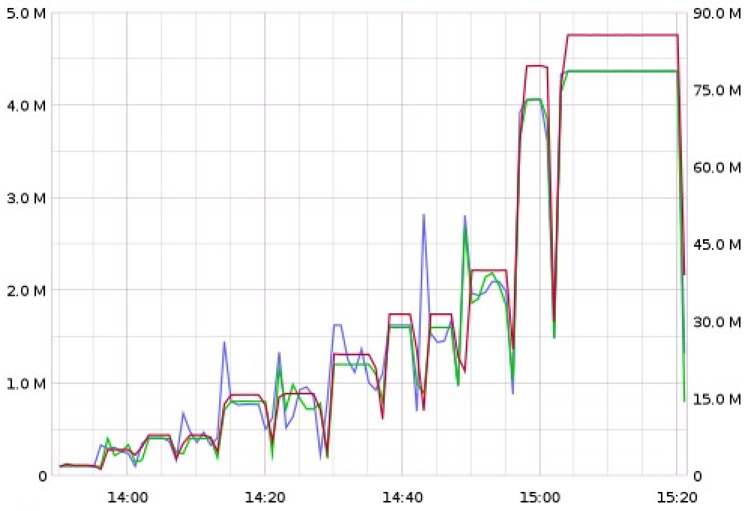

**Figure 4: Graphite visualization**

avoid SLO violations or to closely monitor the application and manually add more resources as needed.

In this scenario, the attendees can experience how Dhalion can automatically react to input load variations and adjust the allocated resources so that the topology remains in a steady state where no backpressure is observed. To demonstrate the flexibility of Dhalion, we have created three autoscaling policies ranging from conservative to aggressive. The first policy, namely the *reactive scaling policy* treats the existence of backpressure as an indication that one or more stages in the topology are under-provisioned and adds more resources by automatically increasing the number of Heron instances in the stage that exhibits backpressure. This policy is described in detail in [7]. As discussed in [7], this policy is quite conservative since it waits for backpressure to appear before taking any action and it only scales one topology stage at a time (the stage that exhibits backpressure). For this reason, in this demo we present two other autoscaling policies that improve on the one published in [7].

The *proactive scaling policy* does not wait for backpressure to appear before adding more resources to a topology stage. In this policy, Dhalion observes the rate at which the size of the input queue of each Heron instance increases. If in any stage the queue sizes are not constant but keep increasing over time, Dhalion determines that the Heron instances of this stage cannot keep up with the input load and characterizes this stage as under-provisioned. Note that this policy does not wait for backpressure to appear but acts much earlier. Finally, the *rapid scaling policy* is the most aggressive one as it does not only scale up the resources of the stage that is the bottleneck but also adds more resources to the downstream stages as well. This is because frequently after scaling up the resources of a stage, the downstream stage gradually becomes the bottleneck as it is not able to handle the increased input load. Thus, it is common to eventually execute multiple scaling operations each one targeting a different stage. The *rapid scaling policy* scales the downstream stages at once, thus reducing the time needed to bring the topology back to a healthy state.

It is worth noting that each policy can result in different reaction time and resource utilization. The attendees will be able to adjust the input load of the topology and observe how the three Dhalion policies react to their changes. We plan to visualize the three policies side-by-side, so that the attendees can compare and contrast them and evaluate their advantages and limitations. Additionally, we plan to demonstrate Dhalion's modularity and extensibility by showing that although the three policies behave differently, they have common building blocks (detectors, diagnosers and resolvers).

## 5. REFERENCES

[1] Apache Heron (incubating). `http://incubator.apache.org/projects/heron.html`.

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

[3] Dhalion Repository. `https://github.com/Microsoft/Dhalion`.

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

[5] Graphite. `http://graphiteapp.org/`.

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

[7] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating Stream Processing in Heron. *PVLDB*, 10(12):1825–1836, 2017.

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

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

[10] A. Toshniwal et al. Storm@Twitter. In *2014 ACM SIGMOD*, pages 147–156.

[11] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[12] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.