Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems

Faria Kalim*†, Thomas Cooper*‡,
Huijun Wu§, Yao Li§, Ning Wang§, Neng Lu§, Maosong Fu§, Xiaoyao Qian§,
Hao Luo§, Da Cheng§, Yaliang Wang§, Fred Dai§, Mainak Ghosh§ and Beinan Wang§
† University of Illinois at Urbana-Champaign, IL, US. Email: kalim2@illinois.edu
‡ Newcastle University, UK. Email: tom.n.cooper@gmail.com
§ Twitter, Inc. Emails: {huijunw, yaoli, nwang, nlu, mfu, xqian, hluo, dac, yaliangw, fdai, mghosh, beinanw}@twitter.com
*Both authors contributed equally to this work.

***Abstract*—Real-time stream processing has become increasingly important in recent years and has led to the development of a multitude of stream processing systems. Given the varying job workloads that characterize stream processing, these systems need to be tuned and adjusted to maintain performance targets in the face of variation in incoming traffic.**

Current auto-scaling systems adopt a series of trials to approach a job's expected performance due to a lack of performance modelling tools. We find that general traffic trends in most jobs lend themselves well to prediction. Based on this premise, we built a system called Caladrius that forecasts the future traffic load of a stream processing job and predicts its processing performance after a proposed change to the parallelism of its operators. Experimental results show that Caladrius is able to estimate a job's throughput performance and CPU load under a given scaling configuration.

***Index Terms*—Stream Processing, Performance Prediction**

I. Introduction

Many use cases for the deluge of data that is pouring into organizations today require real-time processing. Examples of such use cases include internal monitoring jobs that allow engineers to react to service failures before they cascade, jobs that process ad-click rates, and services that identify trending conversations in social networks.

Many distributed stream processing systems (DSPSs) have been developed to cater to this rising demand, that provide high-throughput and low-latency processing of streaming data. For instance, Twitter uses Apache Heron [1], LinkedIn relies on Apache Samza [2] and others use Apache Flink [3]. Usually, DSPSs run stream processing jobs (or topologies) as directed graphs of operators that perform user-defined computation on incoming data packets, called tuples.

These systems generally provide methods to tune their configuration parameters (e.g., parallelism of operators in a topology) to maintain performance despite variations in the incoming workload. However, to our knowledge, all except one of the mainstream DSPSs have built-in auto-scaling support. The exception is Heron's Dhalion framework [4]. Dhalion allows DSPSs to monitor their topologies, recognize symptoms of failures and implement necessary solutions. Usually, Dhalion scales out topology operators to maintain their performance.

In addition to Dhalion, there are several attempts from the research community to create automatic scaling systems for DSPSs. These attempts usually consist of schedulers whose goal is to minimize certain criteria, such as the network distance between operators that communicate large tuples or very high volumes of tuples, or to ensure that no worker nodes are overloaded by operators that require a lot of processing resources [5]–[7]. While the new topology configurations these schedulers produce may be improvements over the original ones, none of these systems assess whether these configurations are actually capable of meeting a performance target or service level objective (SLO) *before* they are deployed.

This lack of performance prediction and evaluation is problematic: it requires the user (or an automated system) to deploy the new topology configuration, wait for it to stabilize and for normal operation to resume, possibly wait for high traffic to arrive and then analyze the metrics to see if the required performance has been met. Depending on the complexity of the topology and the traffic profile, it may take weeks for a production topology to be scaled to the correct configuration.

A performance modelling system that can provide the following benefits is necessary to handle these challenges:

Faster tuning iterations during deployment Auto-scaling systems use performance metrics from real job deployments to make scaling decisions to allow jobs to meet their performance goals. Performance modelling systems that can evaluate a proposed configuration's performance eliminate the need for deployment, thus making each iteration faster. Of course, any modelling system is subject to errors so some re-deployment may be required. However, the frequency and length of the tuning process can be significantly reduced.

Improved scheduler selection A modelling system would allow several different proposed topology configurations to be assessed in parallel. This means that schedulers optimized for different criteria can be compared simultaneously, which helps achieve the best performance without prior knowledge of these different schedulers.

Enabling preemptive scaling A modelling system can accept predictions of future workloads (defined by tuple arrival rate) and trigger preemptive scaling if it finds that a future workload would overwhelm the current topology configuration.

Caladrius[1,2] is a performance modelling service for DSPSs. Its goal is to predict topology performance under varying traffic loads and/or topology configurations. This reduces the time required to tune a topology's configuration for a given incoming traffic load, significantly shortening the *plan* $\rightarrow$ *deploy* $\rightarrow$ *stabilize* $\rightarrow$ *analyze* loop that is currently required to tune a topology. It also enables preemptive scaling before disruptive workloads arrive. Caladrius can be easily extended to provide other analyses of topology configurations and performance.

Caladrius provides a framework to analyze, model and predict various aspects of DSPSs (such as Apache Heron [1] and Storm [8]) and focuses on two key areas:

Traffic The prediction of the incoming workload of a stream processing topology. Caladrius provides interfaces for accessing metrics databases and methods that analyze traffic entering a topology and predict future traffic levels.

System Performance The prediction of a topology's performance under a given traffic load and configuration. This can be broken down into two scenarios:

- **Under varying traffic load** The prediction of how a currently deployed topology configuration will perform under potential future traffic levels.
- **Using a different configuration** The prediction of how a topology will perform under current traffic levels if its configuration is changed.

This paper makes the following contributions:

1) Motivated by challenges that users face, we introduce the notion of DSPSs modelling which enables fast topology tuning and preemptive scaling, and discuss its properties.
2) We present Caladrius, the first performance modelling and evaluation tool for DSPSs (Section III, IV). Caladrius has a modular and extensible architecture and has been tested on Apache Heron.
3) We validate the proposed models and present one use case for performance assessment and prediction for stream processing topologies (Section V).

We discuss related work in Section VI. Finally, we draw conclusions in Section VII. In the next section, we present the background of stream processing systems and topologies.

II. Background

This section presents a brief overview of DSPSs as well as related concepts and terminologies, particularly those belonging to Apache Heron [1], on which Caladrius is tested. Terms are formatted as *italics* in this section and the rest of the paper aligns to the term definitions given here.

[1]This is the Roman name for the legend of the healing bird that takes sickness into itself; the Greek version of this legend is called Dhalion.

[2]Available under an open source licence at https://github.com/twitter/caladrius

A. Topology Perspective: Component Level

A stream processing job or *topology* can be represented as a directed graph of *components* or *operators*. A component is a logical processing unit, defined by the developer, that applies user-defined functions on a stream of incoming data packets, called *tuples*. The edges between connected components represent the data-flow between the computational units. Source components are called *spouts* in Heron terminology; they pull tuples into the topology, typically from sources such as messaging systems like Apache Kafka [9]. Tuples are processed by downstream components called *bolts*.

B. Topology Perspective: Instance Level

The developer specifies how many parallel *instances* there should be for each component: this is called the component's *parallelism*. All instances of the same component are of same resource configuration. The developer also specifies how tuples from each component's instances should be partitioned amongst the downstream component's instances. These partitioning methods are called *stream grouping*. The most common stream grouping is the *shuffle grouping*, where data is partitioned randomly across downstream instances. The second most common is the *fields grouping*, which chooses the downstream instance based on the hash of one or more data fields in the outgoing tuples. There are several other stream grouping types available and users can also define their own, but these are much less common.

C. Throughput Definitions

When studying a topology, there are three granularities: topology, component and instance levels. When we study any of these entities, we define three kinds of throughputs: *source throughput* which is the throughput that the external source (such as a pub-sub system) provides whilst waiting to be processed by the entity; *input throughput* which is the throughput that the entity ingests; *output throughput* which is the summed throughput of the entity's outputs. An entity will become saturated when its input throughput drops below the source throughput and data will begin to accumulate in the external system waiting to be fetched. Note: We use the terms *throughput*, *traffic* and *rate* interchangeably in this paper.

D. System Perspective

A *topology master* is responsible for managing the topology throughout its lifetime and provides a single point of contact for discovering the status of the topology. Each topology is run on a set of containers using an external scheduler, e.g. Twitter uses Aurora [10] for this purpose. Each container consists of one or more *instances*, a *metrics manager* and a *stream manager*, each of which is run as a dedicated JVM process. The instances can also be Python interpreter processes. The metrics manager is responsible for routing metrics to the topology master and/or an external metrics service, while the stream manager is responsible for routing tuples between the topology's containers. Instances process streaming data one tuple at a time and forward the resultant tuples to the

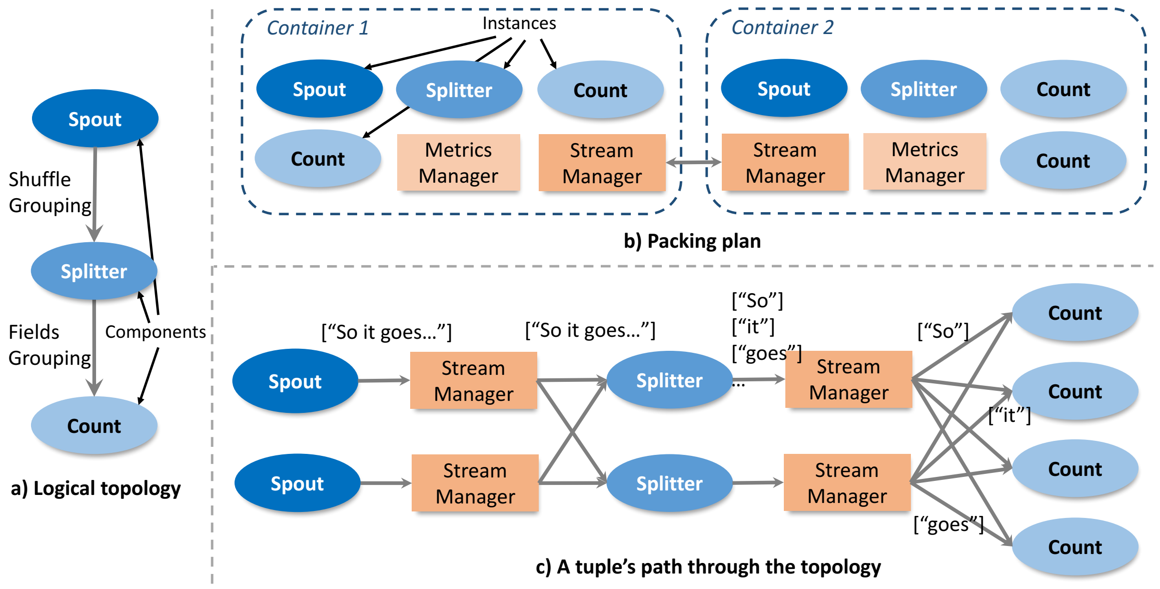

Fig. 1. Sentence-Word-Count Topology: a) A logical DAG of the topology as the developer writes it. b) A physical representation of the topology when it is run. c) A possible path an tuple might take through the topology.

next components in the topology, via the stream manager. Systems that follow similar models include Apache Heron [1], Storm [8], Samza [2] and Flink [3].

E. A Topology Example

Fig. 1 illustrates a sample topology. Fig. 1a) shows the logical representation of the topology. Tuples are ingested into the topology from the spout and passed to the Splitter component which splits the sentences within the tuples into words. The resultant tuples are then passed onto the Counter component that counts the occurrence of each unique word.

Fig. 1b) shows how the topology may look when launched on a Heron cluster. The parallelism of the spout and the Splitter component are both 2 and the parallelism of the Counter bolt is 4. The topology is run in two containers, each of which contains a stream manager for inter-instance communication. This representation of a topology is called its *packing plan*.

Fig. 1c) shows the possible paths a tuple may take through the topology. Though only one path is shown here, there are 16 possible paths through the topology, given the parallelism levels of the components. Stream managers are used for passing tuples between connected instances. If two instances on the same container need to communicate, data will only pass through the local container's stream manager. If the instances run on different containers, the sender's output tuples will first go to its local stream manager, which will then send them to the stream manager on the remote container. The receiving stream manager will be in charge of passing those tuples onto the local receiving instance. Note that this does not increase the number of possible paths in the topology.

III. System Architecture

This section presents a brief overview of Caladrius' architecture, particularly with respect to its interface with Apache Heron [1]. Caladrius consists of three tiers: the API tier, the model tier and the shared model helper tier. These tiers are illustrated in Fig. 2. It is deployed as a web service that can easily be launched in a container and is accessible to developers through a RESTful API provided by the API tier.

A. API Tier

The API tier handles modelling requests from users who would like to use Caladrius to predict future traffic levels and/or the performance of a topology. It is essentially a web server translating and routing user HTTP requests to corresponding modelling interfaces. Caladrius exposes several RESTful endpoints to allow clients to query the various modelling systems it provides. Currently, Caladrius provides topology performance (throughput, backpressure etc.) and traffic (incoming workload) modelling services. Besides performing HTTP request handling, the API tier also fulfills system-wide common shared logistics including configuration management and logging, etc.

It is important to consider that a call to the topology modelling endpoints may incur a wait (up to several seconds, depending on the modelling logic). Therefore, it is prudent to let the API be asynchronous, allowing the client to continue with other operations while the modelling is being processed. Additionally, an asynchronous API allows the server side calculation pipelines to run concurrently.

A response for a call to a RESTful endpoint hosted by the API tier is a JSON formatted string which contains the results of modelling and additional metadata. The type of results listed can vary by model implementation. By default, the endpoint will run all model implementations defined in the configuration and concatenate the results into a single JSON response.

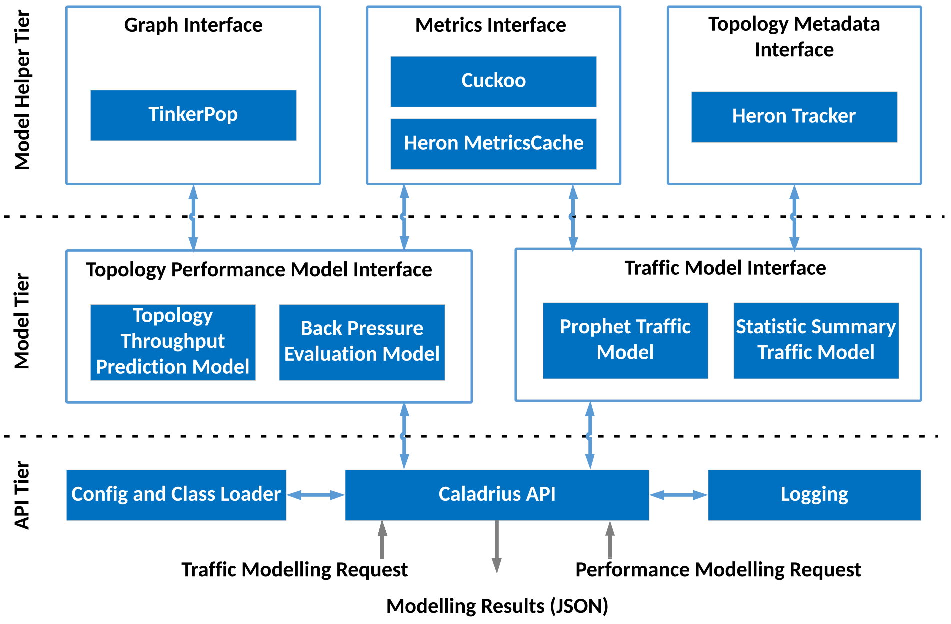

Fig. 2. Caladrius System Overview.

B. Model Tier

The model tier contains both 1) analytical models used to predict topology performance for a given topology packing plan or incoming traffic rate, and 2) predictive models which are used to forecast future source throughput. There are many possible ways to model both the source throughput and the topology performance; thus, Caladrius allows multiple models to be used for each type of analysis. This is illustrated in Fig. 2. The model implementations are configurable through YAML files and the client can specify which models are used when they make requests to Caladrius.

1) Topology Performance: Four golden signals [11] are often used to describe system performance. We define the four golden signals from Heron's perspective:

Latency The latency of a tuple is the time between it entering the topology source, to producing an output result on any sink. A topology's latency is then the maximum of tuple latencies, measured over a period of time.

Traffic The traffic of a topology refers to its *input throughput*, which is the number of tuples it ingests per unit time.

Errors The topology logic may fail a tuple due to some reason in the user-defined processing logic rather than in the Heron framework; this should be handled in the user-defined code or in the topology logic.

Saturation Backpressure is a mechanism whereby a component notifies its upstream sources that it is unable to keep up with the source rate and requires them to stop sending data. This usually occurs if the component's processing rate is unable to keep up with the source rate due to a failed resource or unexpectedly high source rate. In such cases, the component is forced to queue the tuples that it cannot process. When the queue reaches a configured limit, the backpressure signal is triggered and broadcast to all the stream managers in the topology. This results in the spouts not forwarding tuples from the external source to the rest of the topology. Usually, the topology continues to process the queued tuples until it clears the queued backlog, and then requests the sources to restart sending tuples, thus resolving backpressure.

Among the four performance indicators, backpressure is pivotal to Heron. First, it caps the traffic because stopped spouts limit the topology input throughput. Second, backpressure indicates that queues are full and that tuples which are buffered in the queue will experience increased latency.

2) Topology Source Throughput: To ensure that a topology is able to satisfy its performance requirements, it must be configured with sufficient resources and operator parallelism to handle topology source throughput. Caladrius uses Prophet [12], a framework for generalized timeseries modelling to forecast increases in topology source throughput (details in Section IV-A).

C. Model Logistics Tier

Three shared logistics components are often used to supply necessary information to the models — the "metrics" component supplies necessary inputs for the models, and the "graph" and "topology metadata" components hold descriptions of the topologies as state for the models.

1) Graph and Topology Metadata Components: Many analysis techniques for topology performance involve analysis of the topology graph. Caladrius provides a generic graph database interface through which a topology's logical graph (which includes the instances and stream managers) can be uploaded and used for performance analysis. This interface is based on Apache TinkerPop [13], which is an abstraction layer over several popular graph databases and is optimized to perform operations like path calculations. This means that the graph database back-end can be changed if needed (to better serve the requirements of a particular model) without having to re-implement the graph interface code.

A topology's graph is obtained via the Heron Tracker. The Heron Tracker continuously gathers information about Heron topologies running on a cluster, including information about their running status, logical representations and resource allocations, and exposes a RESTful API that can be used to fetch this information.

A topology's logical and physical representation is cached in the graph metadata component. As topology graphs can be large and are usually densely connected, setting up their graphs repeatedly in the database can be time-consuming. In addition, a topology's logical graph changes rarely, albeit the physical representation may be updated. The topology metadata component holds information about the last time the topology is updated. If a change is made to a topology, the information in the graph component is invalidated and updated. Caladrius also provides a graph calculation interface for estimating properties of proposed packing plans.

2) Metrics Provider Component: The Metrics interface provides methods for accessing and summarizing performance metrics from a given metrics source. In the Twitter environment, all metrics that Heron topologies report are gathered by the metrics manager in each container and are stored in Heron MetricsCache and Cuckoo [14], [15] which is Twitter's in house time series metrics database. Concrete implementations of this interface allow metrics to be extracted from Heron MetricsCache or Cuckoo.

These metrics contain information describing the arrival rate of tuples at instances, the number of tuples processed by each instance and the number of tuples emitted per instance etc. In every run, Caladrius pulls these metrics per topology, for traffic prediction and system performance evaluation. Concrete implementations used for the metrics and client interfaces are specified in the configuration file, as are their implementation specific configuration options.

IV. Models: Traffic Forecast and Topology Performance Prediction

As discussed in Section III, Caladrius is a modular and extensible system that allows users to implement their own models to meet their performance estimation requirements. Here, we present a typical use case of Caladrius to evaluate one of the four golden system performance signals — "traffic", and extensively discuss the implementation of the "traffic forecast" and "topology performance prediction" models on top of Heron. Our models can be applied to other DSPSs as long as they employ graph-based stream flow and backpressure-based rate control mechanisms.

A. Traffic Forecast

Caladrius must be able to forecast the topology source throughput, the incoming traffic level into a topology. This is necessary for finding out the topology's performance in the near future. The topology's source throughput is recorded as a time series. This time series consists of tuple counts emitted into the topology per minute.

Time series forecasting is a complex field of research and many methods for predicting future trends from past data exist. For stable traffic profiles with little variation, a simple statistical summary (mean, median, etc.) of a given period of historic data may be sufficient for a reasonable forecast.

However, we find that a large percentage of topologies in the field show strong seasonality. A simple statistical model is not able to predict such strongly seasonal traffic rates. To deal with seasonality, we use more sophisticated modelling techniques. Specifically, we use Facebook's Prophet, a framework for generalized time series modelling [12]. It is based on an additive model where non-linear trends are fit with periodic (yearly, weekly, daily, etc.) seasonality. It is robust to missing data, shifts in the trend, and large outliers.

Caladrius allows users to specify a source time period, on which to base the model, and also whether a single Prophet model should be used for all spouts' source throughput as a whole, or separate models should be created for each spout instance's source throughput. A spout can have many instances, depending on its parallelism. The latter method is slower but more accurate. The user also specifies the future time period over which the source traffic should be forecast. The model then produces various summary statistics for the predicted source rate at the future instances.

B. Topology Performance Prediction

Section IV-A described how we forecast incoming traffic levels for a topology. To make an accurate performance estimation about how the topology will perform at a particular traffic level, we must study the impact of the predicted traffic level on each instance's performance.

1) Modelling Single Instance Throughput: Based on our production experience with Heron, we have the following topology performance observations and we summarize them into assumptions used for the topology performance modelling:

Stream manager and the network are not bottlenecks
The stream manager behaves like a router inside the container. It routes all the traffic for all instances in the container. As the stream manager is hidden from the end user's perspective, it is hard for the end users to reason about it if it forms a bottleneck. Thus, almost all users in the field allocate a large number of containers to their topologies. This means that typically, there are a small number of instances per container and therefore, the stream manager is not a bottleneck. Thus, we assume that the throughput bottleneck is not the stream manager and backpressure is triggered only when the processing speed of the instances is less than their source rate.

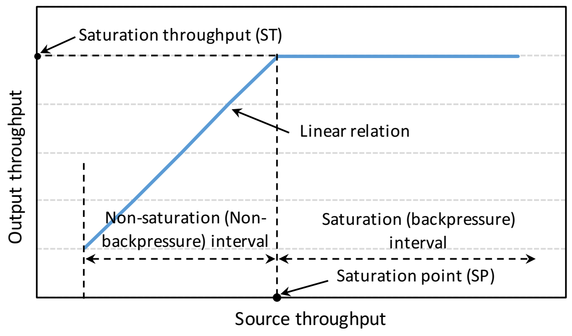

Fig. 3. Performance observation in Heron production experience.

Backpressure is either present or not In Heron, backpressure is triggered by default if the total amount of data pending for processing at one instance exceeds 100Mb (high water mark). Backpressure is resolved if the amount of pending data is below 50Mb (low water mark). Given Twitter's traffic load, small variances can easily push 50Mb of data to instances. This means that although an instance may have reduced the amount of pending data below the lower water mark, enough data is pushed to them that the high water mark is exceeded again. This forces the instance to continue to be in backpressure unnecessarily.

Heron adopts a metric named "backpressure time" to measure how many seconds (in the range from 0 to 60) in a minute when the topology is in backpressure state. Based on the above, "backpressure time" is either close to 60 or is 0, rather than being evenly distributed. Thus, we can approximate the topology's backpressure state to be either 0 or 1.

Based on the assumptions above, we draw the output throughput performance of an instance with a single upstream instance and single downstream instance in Fig. 3. We observe the following features:

Saturation Point (SP) If source traffic rates exceed a threshold, a topology's instances can trigger backpressure. We call this the saturation point (SP) of the instances. When source traffic rises beyond the SP, instances will experience backpressure.

Saturation Throughput (ST) After the source rate exceeds the SP, an instance's output rate reaches and stays at a maximum value, called its saturation throughput (ST). This is because even though source rates are rising, the instance is already pushed to its maximum processing rate.

Linear relation (α) of input and output rates When backpressure is not present and the instance is not saturated, its output rate is proportional to its input rate. It is assumed that this relationship is linear and that its slope represents the input-output (I/O) coefficient (α) of the instance which is determined by its processing logic (together with grouping types if multiple downstream instances exist). Intuitively, $ST = \alpha SP$.

It should be noted that for a large amount of tuples, such as those belonging to Twitter's traffic load, variations in tuple processing rate are insignificant, rendering the processing speed steady and content agnostic.

Given these observations, we express the output rate T_i of a single-input single-output instance i against source rate t_λ as follows:

$$T_i(t_\lambda) = \begin{cases} \alpha_i t_\lambda & : t_\lambda < SP_i \\ ST_i & : t_\lambda \geq SP_i \end{cases} \qquad (1)$$

or simply

$$T_i(t_\lambda) = \min(\alpha_i t_\lambda, ST_i). \qquad (2)$$

The output rate of instances with multiple (m) input streams can be calculated as:

$$T_i(t_\lambda) = \sum_{\lambda=1}^{m} \min(\alpha_i t_\lambda, ST_i) \qquad (3)$$

This approach assumes that input and output streams have a linear relationship, which works well in practice for most topologies. When an instance has only one input, Equation 3 reduces to Equation 2.

Moreover, if there are n outputs, Equation 3 becomes:

$$T_i(t_\lambda) = \sum_{j=1}^{n} T_j(t_\lambda) \qquad (4)$$

$$T_j(t_\lambda) = \sum_{\lambda=1}^{m} \min(\alpha_j t_\lambda, ST_j), j \in [1, 2, \ldots, n] \qquad (5)$$

Where $T_j(t_\lambda)$, α_j and ST_j represent the output rate, the I/O coefficient and the saturation throughput of the j^{th} output stream respectively. α_j is determined by both the instance's processing logic and the type of stream grouping.

2) Modelling Single Component Throughput: Adding the output of all instances of a component gives the component's output. Let's consider a single-input single-output component c first; the multi-input multi-output component's output can be derived from the single-input single-output component in the same way as done for instances in Section IV-B1. Given the component's parallelism p, let the source rate of each instance of the component be $t_{\lambda(1)}, t_{\lambda(2)}, \ldots, t_{\lambda(p)}$. The component's source rate is then:

$$t_\lambda = \sum_{i=1}^{p} t_{\lambda(i)}. \qquad (6)$$

The component output rate is:

$$T_c(p, t_\lambda) = \sum_{i=1}^{p} T_i(t_{\lambda(i)}). \qquad (7)$$

Since a component's instances have the same code, they perform the same functions on incoming tuples. However, the source rate $t_{\lambda(i)}$ to each instance i may not be same due to the upstream grouping type, specified by the user. Here we discuss the impact of the most commonly used grouping types of shuffle (round robin or load balanced) and fields (hash-based or key) grouping.

a) Shuffle Grouped Connections: Shuffle grouped connections between components share output tuples evenly across all downstream instances, leading to:

$$t_{\lambda(1)} = t_{\lambda(2)} = \cdots = t_{\lambda(p)} = \frac{t_\lambda}{p} \tag{8}$$

This means that the routing probability from a source instance to a downstream instance is simply $1/p$, where p is the number of downstream instances, irrespective of the input tuple's content or traffic volume variation over time.

The component output rate is:

$$T_c(p, t_\lambda) = pT_i(\frac{t_\lambda}{p}). \tag{9}$$

Particularly, when $p = 1$, the component has a single instance and Equation 9 reduces to $T_c = T_i$. When $p > 1$, Equation 9 shows that T_c becomes T_i times the number of instances (p).

Consider a component with seasonally varying source rates. We observe several data points of the same parallelism (p) and a range $t_\lambda \in (\eta_1, \eta_2)$ of source throughput. Thus, we can draw a line $T_c(t_\lambda)$ similar to Fig. 3 as long as SP exists in the range (η_1, η_2). This line corresponds to the particular parallelism p. Given this line, we can draw another line of a new parallelism $p' = \gamma p$ by scaling the existing line by γ.

b) Fields Grouped Connections: Fields grouping chooses downstream instances based on the hash of one or more data fields in the tuple and therefore, depending on the values of these fields, may favor particular downstream instances. However, we observed that, in typical Twitter topologies, the volume of tuples and the diversity of keys within them mean that the bias towards certain downstream instances is not strong when averaged over a long-time window. Thus, we assume that the source traffic bias remains unchanged over time in the following discussions.

Below we discuss two changes in a topology's execution:

Varying source traffic load with fixed parallelism By observing the source rate at a particular component parallelism, we can identify whether the data flow is biased towards a subset of instances belonging to downstream components. This allows us to predict the amount of data each downstream instance will receive if the source rate changes. Let the new overall source rate be $t'_\lambda = \beta t_\lambda$. With the steady data set bias assumption, traffic distribution is measured along time and is distributed across all p instances of the operator. Thus, we have:

$$t'_\lambda = \beta t_\lambda = \sum_{i=1}^{p} \beta t_{\lambda(i)} \tag{10}$$

We can calculate the component output rate under a different source traffic load (Equation 11). We observe that the output rate of each instance is proportional to the original one by β when its new source traffic load falls into the linear interval, and reaches the ST otherwise.

$$\begin{aligned} T_c(p, t'_\lambda) &= T_c(p, \beta t_\lambda) \\ &= \sum_{i=1}^{p} T_i(\beta t_{\lambda(i)}) \\ &= \sum_{i=1}^{p} \min\left(\beta T_i(t_{\lambda(i)}), ST_i\right) \end{aligned} \tag{11}$$

Varying parallelism with fixed source traffic load Changing parallelism can affect how tuples are distributed among instances when using fields grouping. This complicates the calculation of the routing probability for fields grouped connections under a different parallelism. The routing probability describes the likelihood that a tuple will pass from a particular instance to another and is a function of the data in the tuple stream and their relative frequency. Therefore, the proportion of tuples that go to each downstream instance depends entirely on the nature of the data contained in the tuples.
Fields grouping chooses an instance to send data to by taking the modulo of the hash value of the relevant tuple field(s) against the number of parallel instances. The modulo operation cannot be reversed, making it impossible to predict routing in a new packing plan. However, we found in some cases that the data set distribution is uniform or load-balanced in a large data sample set. Under this circumstance, the component behaves as Equation 9. A potential solution for a biased data set is that a user can implement their own customized key grouping to make the traffic distribution predictable and plug the corresponding model into Caladrius.

3) Modelling Topology Throughput: A topology is a directed graph, which can contain one or more critical paths. A critical path is the path which limits the entire topology's throughput. Once the model for each component is built, the throughput performance on the critical path can be evaluated. We assume that there are N components on the critical path and the source throughput (t_0) is known, either via the measured actual throughout or the forecasted as described in Section IV-A. The user specifies the parallelism configuration for each component to be $\{p_1, p_2, \ldots, p_N\}$. The output throughput of the critical path (t_{cp}) can be calculated by chaining Equation 7:

$$t_{cp} = \underbrace{T_{c(N)}(p_N, T_{c(N-1)}(\ldots T_{c(2)}(p_2, T_{c(1)}}_{\text{N}}(p_1, t_0))) \ldots) \tag{12}$$

Once we have t_{cp}, we can trace backwards and find the saturation point of the topology:

$$t'_0 = \underbrace{T^{-1}_{c(1)}(T^{-1}_{c(2)}(\dots T^{-1}_{c(N-1)}(T^{-1}_{c(N)}}_{\text{N}}(t_{cp}))) \dots) \quad (13)$$

Moreover, we can identify if there is or will potentially be backpressure by comparing t_0 and t'_0:

$$\text{risk}_{\text{backpressure}} = \begin{cases} low & : t'_0 < t_0 \\ high & : t'_0 \sim t_0 \end{cases} \quad (14)$$

We can also locate the component or instance with high backpressure risk while creating the chain in Equation 12.

For some topologies, the critical path cannot be identified easily. Under this situation, multiple sub-critical path candidates can be considered and predicted at the same time. The critical path selection problem is out of the scope of this paper.

V. Experimental Evaluation

As we depend on the Prophet library for the topology source traffic forecast, the performance evaluation of Caladrius' traffic prediction will not be discussed here. We focus on the evaluation of the topology performance prediction model and its integration into Caladrius as an end-to-end system. The evaluation is conducted in two main parts.

1) We evaluate the output rate predictions. We validate our observation and assumptions for the single instance in Section V-B, and the models for the single component in Section V-C and the critical path in Section V-D.
   DSPSs usually provide scaling commands to update the parallelism of their components. For example, Heron provides an update command to alter a component's parallelism. Although users have tools to scale topologies, it is hard to predict changes in performance after running the commands. Some existing systems, such as Dhalion, use several scaling rounds to converge on the users' expected throughput SLO, which is a time-consuming process.
   Conversely, Caladrius can predict the expected throughput given a new set of component parallelisms, which gives users useful insights on how to tune their topologies. This can be done by executing the update command in *dry run* mode. It should be noted that in dry run mode, the new packing plan and the expected throughput is calculated without requiring topology deployment, thus significantly reducing the time taken to find a packing plan to satisfy the SLO.
2) Besides throughput, we also conduct CPU load estimation for updated parallelism levels in Section V-E. The CPU load primarily relates to the processing throughput, which makes its prediction feasible once we have a throughput prediction.

A. Experimental Setup

Previous work on DSPSs [4] used a typical 3-stage Word Count topology to evaluate the systems under consideration. In our work, we use the same topology, shown in Fig. 1-a. In this topology, the spout reads a line in from the fictional work *The Great Gatsby* as a sentence and emits it. The spouts distribute the sentences to the Splitter component using shuffle grouping. The Splitter component splits the sentences into words that are then forwarded to the Counter component via a fields grouped connection. Finally, the Counter component counts the number of times each word has been encountered.

As there is no external data source in the experiments, we use a special kind of spout whose output rate matches the configured throughput if there is no backpressure triggered by the topology instances, and their throughput is reduced if backpressure is triggered. Unless mentioned otherwise, the spout's parallelism in each experiment is set to 8.

We run the topology on Aurora, a shared cluster with Linux cgroups isolation. The topology resource allocation is calculated by Heron's round-robin packing algorithm — 1 CPU core and 2GB RAM per instance, without disk involved in the evaluation. Note that the evaluation topology was constructed primarily for this empirical evaluation, and should not be construed as being the representative topology for Heron workloads at Twitter.

We use output throughput as the evaluation metric in our experiments. We note that the output rate of a spout or bolt is defined as the number of tuples generated per minute.

We tune the Word Count topology to perform in ways that we expect in production settings i.e., there are no out-of-memory crashes, or any other failure due to resource starvation during scheduling or long repetitive garbage collection cycles. The experiments were allowed to run for several hours to attain steady state before measurements were retrieved.

B. Single Instance Model Validation

To validate the single instance model in Fig. 3, we set the Splitter component's parallelism to 1. The topology spout instances were configured to have an output traffic rate of from 1 to 20 million tuples per minute with an additional step of 1 million tuples per minute. Meanwhile, the parallelism of the Counter component is set to 3 to prevent it from becoming a bottleneck. We collect the Splitter processed-count and emit-count metrics as they represent the instance's input and output rates. The observation was repeated 10 times and the throughput with 90% confidence intervals is drawn in Fig. 4.

There are two series of measurements of the Splitter instance in Fig. 4. One is the input rate and the other is output rate. The x-axis is the spout's output rate, and the y-axis shows the two series value in units of a million tuples per minute. We can see the two series increase until approximately 11 million tuples per minute, which is the SP. After the SP, both series tend to be steady, among which the output rate is the ST.

Fig. 5 shows the ratio of output rate over input rate, which is between 7.63 and 7.64 — as the variation of output rate with respect to source rate is negligible, it can be roughly treated as a constant value. The slope represents the number of words in a sentence and so the average of these represents the average length of a sentence in *The Great Gatsby*.

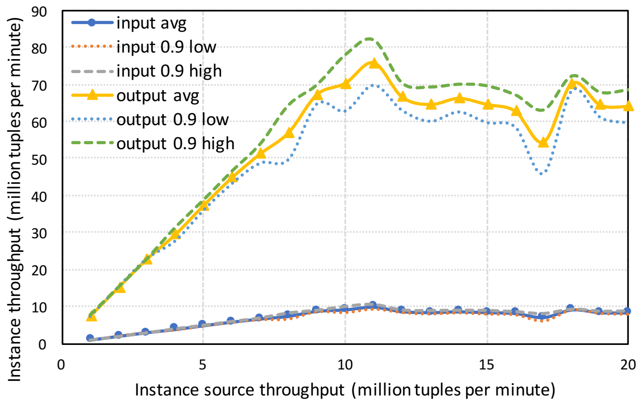

Fig. 4. Instance throughput (output and input) vs. topology source throughput.

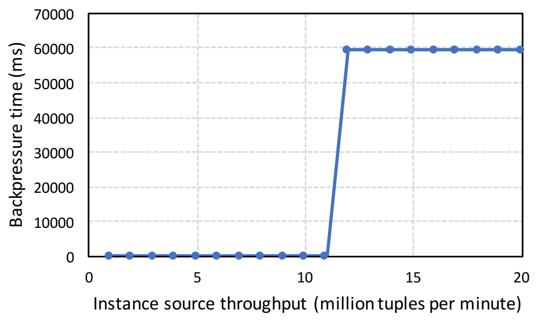

Fig. 6. Instance backpressure time vs. instance source throughput.

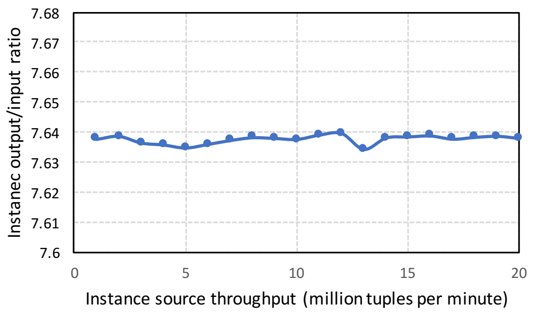

Fig. 5. Instance output/input ratio vs. instance source throughput.

We note how the trace in Fig. 5 fluctuates in the non-saturation interval, which is possibly due to competition for resources within the instances. An instance contains two threads: a gateway thread that exchanges data with the stream manager and a worker thread that performs the user defined logic. When the input rate increases, the burden on the instance gateway thread and communication queues increases, which results in less resources allocated to the processing thread. However, the performance degradation in the processing thread is small and transient.

Time spent in backpressure is presented in Fig. 6. We observe that backpressure occurs when the source throughput reaches around 11 million (the SP identified earlier). The time spend in backpressure rises steeply from 0 to around 60000 milliseconds (1 minute) after it is triggered.

From the observation above, we note that to draw the curve in Fig. 3 for a given instance, we need at least two data points: one in the non-saturation interval and one in the saturation interval. We can get these points from two experiments: one without and one with backpressure.

C. Single Component Model Validation

When we observe running topologies in a data center, we see that source traffic varies with time and may have the same throughput value at multiple points. This means that we can observe multiple instances of a particular source traffic rate. In the experiments we emulate multiple observations of the same source rate by restarting the topology and observing its throughput multiple times.

To validate the single component model, we follow the previous single instance evaluation: we focus on the Splitter component and start with a parallelism of 3 as in Fig. 7.

We can see that the throughput lines of a component have similar shape to those of the instances shown in Fig. 4, but scaled according to the parallelism. The total source rate into the instance ranges from 2 to 68 million tuples per minute, and the SP is around 30 million. The piecewise linear regression lines are also marked as dash lines, and the output over input ratio is calculated to be 7.638, which is consistent with the result in Fig. 5.

Based on the observation of the Splitter component with a parallelism of 3, we can predict its throughput with another parallelism level. Given the discussions of Equation 9, we plotted the predictions of throughput with parallelism 2 and 4, as dashed lines for both input and output, in Fig. 7. The predicted input and output inflection points with a parallelism of 2 are around 18 million and 140 million respectively, while those for parallelism of 4 are 36 million and 280 million.

To evaluate the prediction, we deployed the topology with Splitter component parallelism to be 2 and 4, and measured both input and output rates as shown in Fig. 8. In the non-backpressure interval, the predicted curves match the measured ones. The ST prediction error, which is defined as the difference between the corresponding predicted and observed regression lines over the observed regression line of output rate ($|ST_{\text{prediction}} - ST_{\text{observation}}|/ST_{\text{observation}}$), is around $(140 - 136)/136 = 2.9\%$ for parallelism 2 and $(287 - 280)/280 = 2.5\%$ for parallelism 4. We can see that the ST predictions of parallelism 2 and 4 match well with the measured ones, with acceptable small variations.

D. Critical Path Model Validation and Topology Throughput Prediction

For the example topology in Fig. 1, the critical path is the only path going through the three components. Since our spout does not ingest data, we assume its source, input and output

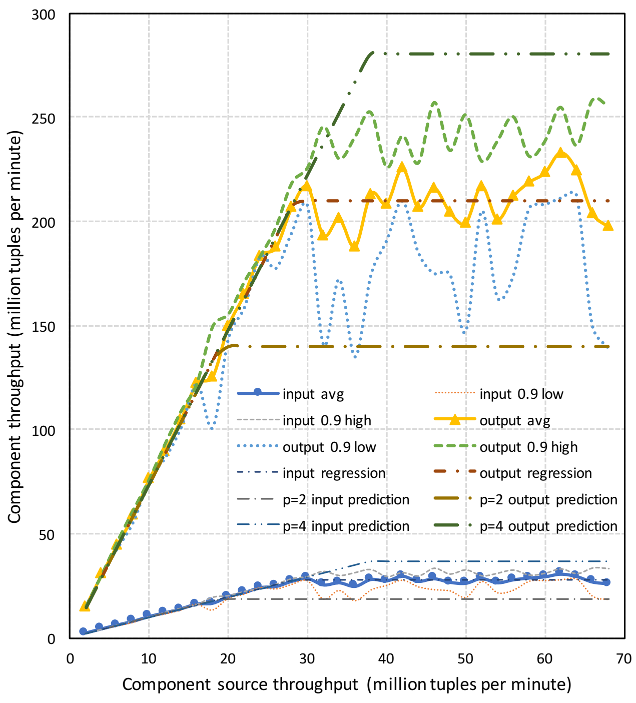

Fig. 7. Component (Splitter) throughput measurements of parallelism 3 and predictions of parallelism 2 and 4.

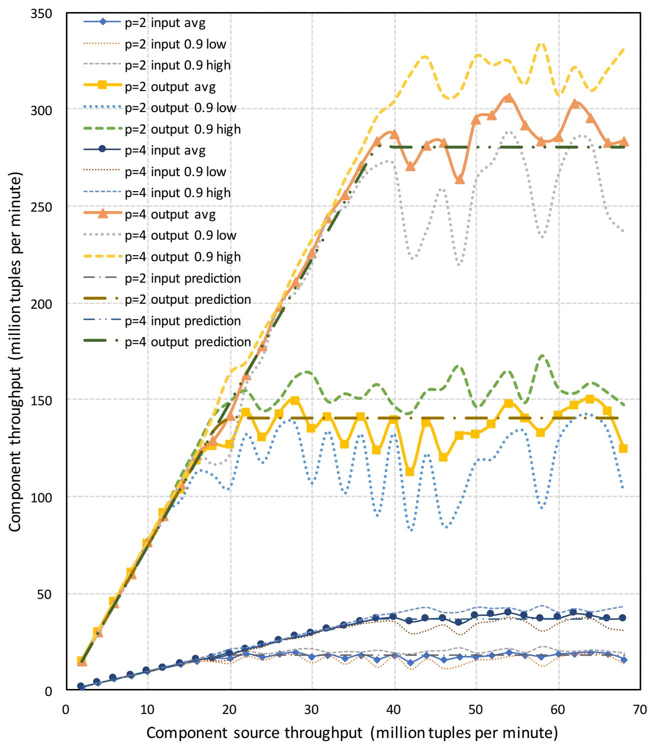

Fig. 8. Validation of component (Splitter) throughput prediction of parallelisms of 2 and 4.

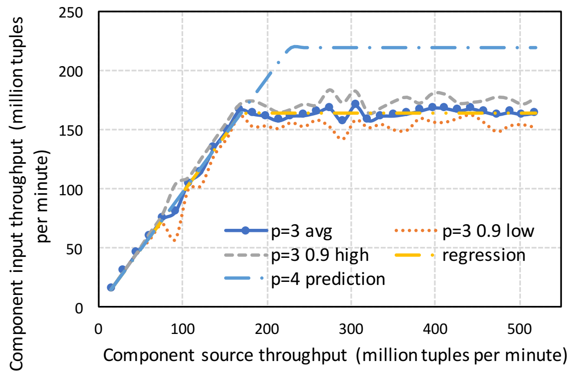

Fig. 9. Component (Counter) input throughput: observation and prediction.

throughput are same. In the previous experiments, we have built a model for the Splitter component. We did the same for the Counter component and show its model in Fig. 9. Moreover, we observed the test dataset is unbiased fortunately, thus we use Equation 9 for the sink bolt.

Now we have all the three component models on the critical path, and we can predict the critical path throughput by applying Equation 12. We choose the parallelisms in Fig. 1. The predicted topology output throughput (the sink bolt's processing throughput) is shown in Fig. 10. We deployed a topology with the same parallelism in our data center, and measured its output throughput, also shown in Fig. 10. This figure shows the observation matches the prediction with an error of $(139 - 135)/139 = 2.8\%$.

E. Use Case: Predict CPU load

The logic executed by a component's instances can be categorized as CPU-intensive or memory-intensive, whose CPU or memory load can be predicted. Input rate also significantly impacts an instance's resource usage. Two factors are worth considering while performing our micro-benchmarks:

1) The saturation state i.e., whether a component triggers backpressure. When the component triggers backpressure, its CPU or memory load is supposed to be at the maximum possible level as the processing throughput of its instances also reaches their maximum points.
2) The resource limits of the containers that run the instances (especially in terms of memory). Instances may exceed the container memory limit when their input rate rises to sufficiently high levels, which is rare in a well-tuned production job but can still happen.

In this section, we choose the CPU load of instances as an example. We observed that the CPU usage is linearly related to the input rate per instance. Once we have the observation of several data points of {CPU load, input rates, source rates}, we can prepare two intermediate results:

- We can depict the throughput model {input rates, source rates}, as we did in the previous evaluations.
- We can then use the model {CPU load, input rates}

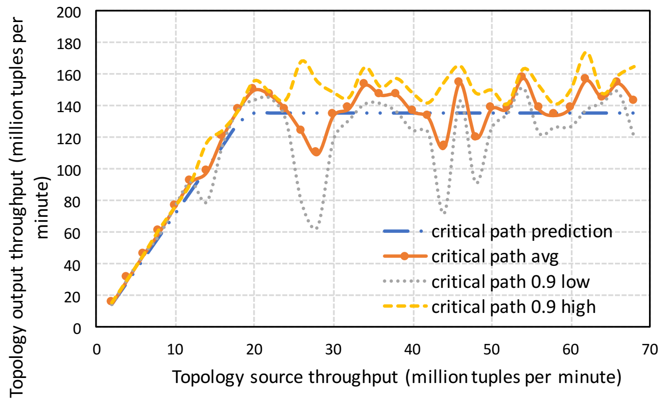

Fig. 10. Topology predicted and measured output throughput.

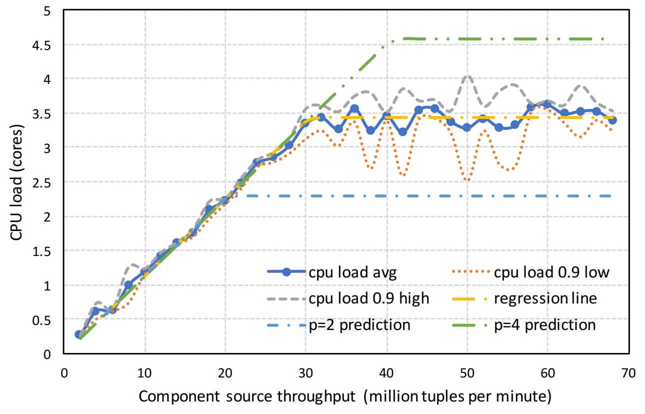

Fig. 11. Observed and predicted CPU load of Splitter component.

to calculate the linear ratio or the slope ψ = CPU load/input rate.

Given the target source rates, we use our model {input rates, source rates}, to find estimated input rates. We then amplify the input rates by ψ to estimate the CPU load.

We set the parallelism level of the Splitter component to 3 and observe the CPU load of its instances in Fig. 11. The CPU load is collected from the Heron JVM native metrics, which presents the recent CPU usage for the Heron instance's JVM process. The predicted CPU load regression lines are shown in the same figure as dashed lines for parallelisms 2 and 4 for the Splitter component.

Additionally, we configured the source rate and measured its CPU load for parallelisms 2 and 4 for the Splitter component. Fig. 12 shows the measured CPU load vs. the predicted values. The prediction error is $(2.399 - 2.284)/2.399 = 4.8\%$ for parallelism 2 and $(4.568 - 4.435)/4.435 = 3\%$ for parallelism 4, which is higher than the output rate prediction error. This is because error has accumulated for the chained prediction steps.

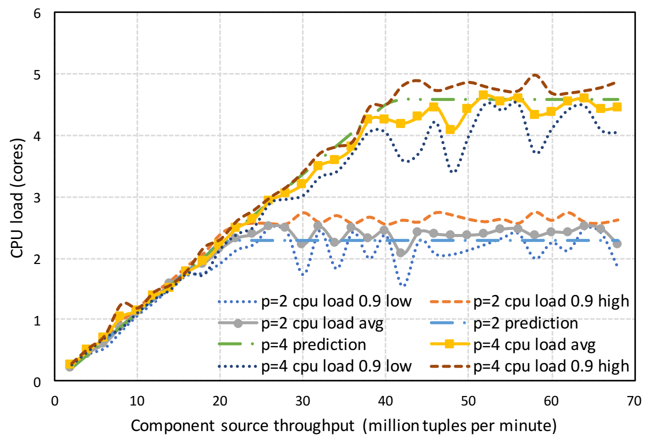

Fig. 12. Validation of the CPU load prediction of Splitter Component.

VI. Related Work

A. *Performance Prediction*

Traffic prediction is used for performance improvements in several areas. For instance, the ability to predict traffic in video streaming services can significantly improve the effectiveness of numerous management tasks, including dynamic bandwidth allocation and congestion control. Authors of [16] use a neural network-based approach for video traffic prediction and show that the prediction performance and robustness of neural network predictors can be significantly improved through multi-resolution learning.

Similarly, several predictors exist in the area of communication networks such as ARIMA, FARIMA, ANN and wavelet-based predictors [17]. Such prediction methods are used for efficient bandwidth allocation (e.g., in [18]) to facilitate statistical multiplexing among the local network traffic.

However, the work done on traffic prediction in DSPSs is limited. Authors of [19] perform traffic monitoring and re-compute scheduling plans in an online manner to redistribute Storm topology workers to minimize internode traffic.

Authors of [20] proposed a predictive scheduling framework to enable fast, distributed stream processing, which features topology-aware modelling for performance prediction and predictive scheduling. They presented a topology-aware method to accurately predict the average tuple processing time of an application for a given scheduling solution, according to the topology of the application graph and runtime statistics. They then present an effective algorithm to assign threads to machines under the guidance of prediction results.

B. *Resource Management in Stream Processing*

Topology scheduling in DSPSs is a thoroughly investigated problem. Borealis [21], a seminal DSPS, proposed a Quality of Service (QoS) model that allows every message or tuple in the system to be supplemented with a vector of metrics which included performance-related properties. Borealis would inspect the metrics per message to calculate if the topology's QoS requirements are being met. To ensure that these guarantees are met, Borealis would balance load to use slack resources for overloaded operators. On the other hand, STREAM [22] is a DSPS that copes with a high data rate by providing approximate answers when resources are limited.

Authors of [23] focus on optimal operator placement in a network to improve network utilization, reduce topology latency and enable dynamic optimization. Authors of [24] treat stream processing topologies as queries that arrive at real-time and must be scheduled subsequently on a shared cluster. They assign operators of the topologies to free workers in the cluster with minimum graph partitioning cost (in terms of network usage) to keep the system stable.

A plethora of work [4], [25], [26] exists that gathers metrics from physically deployed topologies to find resource bottlenecks and scales topologies out in multiple rounds to improve performance. A relatively new topic for DSPSs is using performance prediction for proactive scaling and scheduling of tasks; Caladrius takes a step in this direction.

VII. Conclusion

In this paper, we described a novel system developed at Twitter called Caladrius, that models performance for distributed stream processing systems and is currently integrated with Apache Heron. We presented Caladrius' system architecture, three models for predicting throughput for a given source rate, and one use case for CPU load prediction. We illustrated the effectiveness of Caladrius by validating the accuracy of our models and Caladrius' prediction of throughput and CPU load when changing component parallelism.

Acknowledgements

This work was carried out by Thomas and Faria during their internships at Twitter. Thomas' work is partially based on research supported by the Engineering and Physical Sciences Research Council, Center for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1]. He would like to thank Dr. Paul Ezhilchelvan (paul.ezhilchelvan@ncl.ac.uk) for his assistance. Faria's work is partially supported by the grant NSF CNS 1409416. She would like to thank Prof. Indranil Gupta (indy@illinois.edu) for his assistance.

References

[1] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[2] M. Kleppmann and J. Kreps, "Kafka, samza and the unix philosophy of distributed data." *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 4–14, 2015.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: self-regulating stream processing in heron," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1825–1836, 2017.

[5] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2014, pp. 296–302.

[6] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel & Distributed Systems*, no. 1, pp. 1–1, 2014.

[7] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 13–22.

[8] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[9] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[10] "Apache Aurora," http://aurora.incubator.apache.org, 2014, [Online; Accessed February 22, 2019].

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[12] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, no. 1, pp. 37–45, 2018.

[13] "Apache TinkerPop," http://tinkerpop.apache.org/, 2016, [Online; Accessed February 22, 2019].

[14] @anthonyjasta, "Observability at twitter: technical overview, part i," https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html, 2016, [Online; Accessed February 22, 2019].

[15] ——, "Observability at twitter: technical overview, part ii," https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-ii.html, 2016, [Online; Accessed February 22, 2019].

[16] Y. Liang, "Real-time vbr video traffic prediction for dynamic bandwidth allocation," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 34, no. 1, pp. 32–47, 2004.

[17] H. Feng and Y. Shu, "Study on network traffic prediction techniques," in *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, vol. 2. IEEE, 2005, pp. 1041–1044.

[18] Y. Luo and N. Ansari, "Limited sharing with traffic prediction for dynamic bandwidth allocation and qos provisioning over ethernet passive optical networks," *Journal of Optical Networking*, vol. 4, no. 9, pp. 561–572, 2005.

[19] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2014, pp. 535–544.

[20] T. Li, J. Tang, and J. Xu, "Performance modeling and predictive scheduling for distributed stream data processing," *IEEE Transactions on Big Data*, vol. 2, no. 4, pp. 353–364, 2016.

[21] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine." in *Cidr*, vol. 5, no. 2005, 2005, pp. 277–289.

[22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, approximation, and resource management in a data stream management system." in *CIDR*, 2003, pp. 245–256.

[23] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proceedings of the 22nd International Conference on Data Engineering*. IEEE, 2006, pp. 49–49.

[24] J. Ghaderi, S. Shakkottai, and R. Srikant, "Scheduling storms and streams in the cloud," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1. ACM, 2015, pp. 439–440.

[25] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven Multi-Tenant Stream Processing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 249–262. [Online]. Available: http://doi.acm.org/10.1145/3267809.3267832

[26] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 22–31.