Utrecht University

channable

Faculty of Science
Computing Science
Master's Thesis

April 2023

Scheduling data feed processing jobs

Supervisors:
dr. H.H. Liu
dr. W.S. Swierstra
R. Kreuzer (daily supervisor)

Student:
Maarten van den Berg
5636450

Abstract

Online advertising is an important strategy for companies that sell products on the internet to find customers. These companies often make use of an e-commerce system, which stores data on the products that the company sells. If a company's inventory changes often, it may be desirable to automate the process of creating and updating advertisements, to reduce the workload of keeping advertisements up to date.

Channable is a company that provides a tool for automated creation of advertisements, based on the data in a company's e-commerce system. Channable offers a product feed processing system which connects to a company's e-commerce system and regularly downloads information on the company's inventory. Once this data has been downloaded the system can apply customer-defined processing rules to the data and convert the data to a format suitable for submission to one or more advertising platforms or marketplaces.

The heavy computational lifting in this system is performed by rule processing servers, which accept inventory data and customer-defined processing rules and produce a datastream that has been processed according to the customer-defined rules. Channable uses multiple of these rule processing servers for redundancy and performance reasons, and so it must balance the workload between the servers. The current method for assigning work to the servers uses a distributed scheduler. This scheduler has some limitations which cause it to distribute the work unevenly between servers, causing some servers to regularly become overloaded while other servers sit idle.

In this thesis we implement a better method for assigning work to the rule processing servers, by making use of a centralised scheduler. We compare two methods for detecting overloaded servers and one alternative algorithm for assigning work to servers to the current approach by performing experiments using real-world data to determine which scheduling approach performs best.

We show that using our scheduler can significantly improve the performance of the rule processing system: our best-performing scheduling algorithm speeds up the average duration of low-priority jobs by a factor of 2.2 and reduces the average waiting time of low-priority jobs by a factor of 3.5. We are also able to significantly reduce the variance in waiting time between the different rule processing servers, making the product feed processing system's performance more predictable.

Contents

1 Introduction

Online advertising is an important tool for companies to attract new customers, especially for companies that sell products on the internet. Companies selling products on the internet generally use an *e-commerce platform* to make their products available to customers. E-commerce platforms store data on the products that the company sells (called *product data*) and present a storefront to users that allows users to purchase the products.

Companies may also want to advertise their products on different platforms outside their own e-commerce platform. Many platforms allow creating advertisements or product listings automatically, by supplying the product data in a format defined by the external platform. These formats can differ from the formats supported natively by a company's e-commerce platform, requiring the use of additional software to make the product data adhere to a target platform's requirements.

Channable provides a *product feed processing system* that can perform this transformation automatically. This system's purpose is to take in product data as produced by a client's e-commerce platform, convert the product data to an intermediate format that allows platform-independent manipulation of the product data, and then automatically convert it into a format suitable for submission to a marketplace or advertisement provider. The system is provided as a Software-as-a-Service-tool, which means that Channable hosts the system and processes data for many customers through the same infrastructure.

Channable's product feed processing system also allows customers to automatically modify the imported product data while it is being processed, by allowing its customers to define *processing rules*. These processing rules can for example be used to correct mistakes in the input data, or to prevent certain products from being advertised. The rules are applied to the product data by a special class of servers in the product feed processing system called *rule processing servers*. These servers have more computational resources compared to the other servers in Channable's system, and provide a centralized computing service for the relatively less-powerful worker servers that handle communication with external services.

The product feed processing system handles many processing requests: as of Q3 2022 the job scheduler executes over 1.3 million jobs per day, processing product data on over two billion products. The rule processing servers handle about 2.8 million rule processing tasks per day. In order to be able to process this much data the product feed processing system needs to divide the workload over the available rule processing servers.

Channable's production system currently uses a decentralized scheduling approach, where each worker server can decide independently on which rule processing server a rule processing task should run. This scheduling method unfortunately has some shortcomings which cause the workload to be distributed unevenly. The uneven workload distribution causes problems in Channable's production environment: at peak moments it is often the case that some servers become overloaded, while other servers are nearly idle.

In this thesis project we seek to address these problems by implementing a better method for assigning work to the rule processing servers. We aim to answer the following research question:

> *How to dynamically schedule rule processing jobs to evenly divide the workload among a number of rule processing servers?*

Outline

This thesis is divided into two parts.

In part I we further introduce the context for our thesis project. In section 2 we describe the product feed processing system in more detail, and in section 3 we describe the rule processing service,

the component of the product feed processing system that our project focuses on. In section 4 we outline the scheduling problem that we aim to solve in this thesis project, and in section 5 we describe found literature that is relevant to this scheduling problem.

In part II we describe the steps we took to investigate how we could improve the system's performance. In section 6 we define our research question and optimization goals and describe the setup of our experiments. In section 7 we describe the environment in which we run our experiments, and in 8 we describe the process of capturing the data we use from the production environment. In section 9 we describe the scheduling approaches that we evaluate in this project. In section 10 we present the results of our experiments and compare the performance of the production environment against the performance of our experiment environment under our scheduling policies.

In section 11 we present our conclusions and in section 12 we suggest possibilities for future work.

Part I
Context

2 The product feed processing system

In this section we describe the product feed processing system we introduced in section 1 and the components of this system that are relevant for our thesis project.

2.1 Phases

The product feed processing system operates in three *phases*:

- The *import phase*:
  During the import phase the system downloads product data from the customer's e-commerce system. The downloaded product data is converted into an intermediate format and stored in a central database for further processing.
- The *rule processing phase*:
  During the rule processing phase the system applies customer-defined *processing rules* (also simply called *rules*) to the imported set of product data. Rules can either modify items in the imported product data or remove items from the product data, excluding them from further processing.

  The rule processing phase is *pure*: applying the same set of rules to the same imported product data will result in the exact same result. This property allows the result of the rule processing phase to be cached.

  The result of the rule processing phase is a filtered set of product data. This product data is still stored in an intermediate format not directly usable by external systems.

  The rule processing phase is described in more detail in section 3.
- The *export phase*:
  During the export phase the system converts the filtered product data to the format accepted by the third-party system that the product data is to be submitted to.

  The result of the rule processing phase is either a text file that is made available for download, or a set of calls to an export platform's API to publish the processed product data.

2.2 Data organization

The data in the product feed processing system is organized into *projects*. Each project contains a configuration of import channels, processing rules and export channels, which all operate on the same product data.

Projects are independent from each other: any change to a project's import channel, export channel or processing rule only affects the product data inside that project.

2.3 System triggers and priority

The product feed processing system can be triggered either manually (in response to a user clicking a button) or automatically (according to a customer-defined schedule).

Manual processing requests are prioritised over automated processing, to improve user experience in Channable's web application. This is done because the results of manual processing requests will need to be shown to the user, so by prioritizing manual requests they can complete sooner.

The prioritisation is done by assigning each request a *priority*, which is either 'high' (for manual requests) or 'low' (for automated requests).

The effect of marking a request as 'high priority' varies by component, but generally results in processing for any low-priority requests being paused as long as there are high priority requests to handle.

2.4 Components

The product feed processing system consists of various components which together provide the functionality of the system. An overview of the components of the product feed processing system is shown in figure 1.

Only the components shown in bold are relevant for this project's scheduling problem. A description of these components follows:

The web backend handles incoming HTTP requests from customers after they have passed through the webserver and load-balancer services. It is implemented as a custom-built web service using the Python programming language.

The web backend creates/updates the configuration for the product feed processing system by storing the appropriate configuration (import channels, processing rules, export channels, schedule) in the main database.

To answer incoming HTTP requests the web backend may issue requests to the rule processing servers (e.g. to inspect the results of applying the processing rules) or to the job scheduler (to trigger a high-priority job).

The worker servers execute the import jobs and export jobs submitted to the job scheduler.

The worker servers receive job specifications from the job scheduler once the scheduler determines a job can be run on a worker server. Job specifications specify which import or export phase to run, for which project the job is to be run, and the job's priority. The worker server will then start a Linux process that executes the job according to the received job request, monitor its progress, and eventually report success or failure to the job scheduler.

During the execution of an export job (i.e. a job which runs the export phase) a worker server will offload the process of applying the configured processing rules to one of the available rule processing servers.

The rule processing servers each host an instance of the *rule processing service*. The rule processing service accepts HTTP requests from both the worker servers and the web backend. It is implemented as a custom-built web service using the Haskell programming language.

The rule processing requests specify which project's product data is being requested and contain all rules that must be applied to the product data. The rule processing service will then retrieve the appropriate product data from the item store, apply the configured rules, and then stream the results back to the requesting service.

The rule processing service is the main focus of this thesis and is described in more detail in section 3.

The item store stores the imported product data. During the import phase the worker servers store the imported items in the item store, then during the rule processing phase the rule processing servers read the items from the item store.

Channable is in the process of migrating the item store from an instance of the PostgreSQL database system to Google Cloud Storage (GCS), an object storage system. This migration consists of removing the item data from the PostgreSQL database and storing it in a file in GCS instead, replacing the data in the PostgreSQL with a marker indicating how to retrieve the data from GCS.

Due to this migration the stored product data is currently split between these systems, though this split is largely invisible to all components except for the rule processing service. In this thesis we will refer to the product data as a whole as *the item store*. Where references to a specific storage backend are needed we will refer to the PostgreSQL database as the *item store database*, and to the Google Cloud Storage environment as the *item store objects*.

In addition to the components listed above, there are also two support services that are not specific to the product feed processing system.

The service discovery system is used by all components to detect on which servers the other components are running. The service discovery system is an instance of the Consul service discovery system and is available on every server.

Each component registers itself with the service discovery system on startup, after which the upstream services get notified that a new instance of a service is available. Any component can query the service discovery system to determine which servers are available to handle a request.

The service discovery system will perform periodic health checks to verify whether each component is still able to accept processing requests. When a component is found to be unhealthy (e.g. crashed) the service discovery system will mark it as 'unhealthy', which causes upstream components to stop sending requests to the component until it recovers.

The monitoring system collects metrics on each of the components of the product feed processing system. The monitoring system is an instance of the Prometheus monitoring system installed on a dedicated monitoring server.

Each component in the product feed processing system exposes a HTTP API with a 'metrics endpoint', which exposes time-series data on the component's performance. The monitoring system periodically requests this data and saves it in a central time-series database so the data can be queried.

The monitoring system also provides an alerting function which is used to send alerts to Channable's operations team if a component is unhealthy for a long amount of time or under high load.

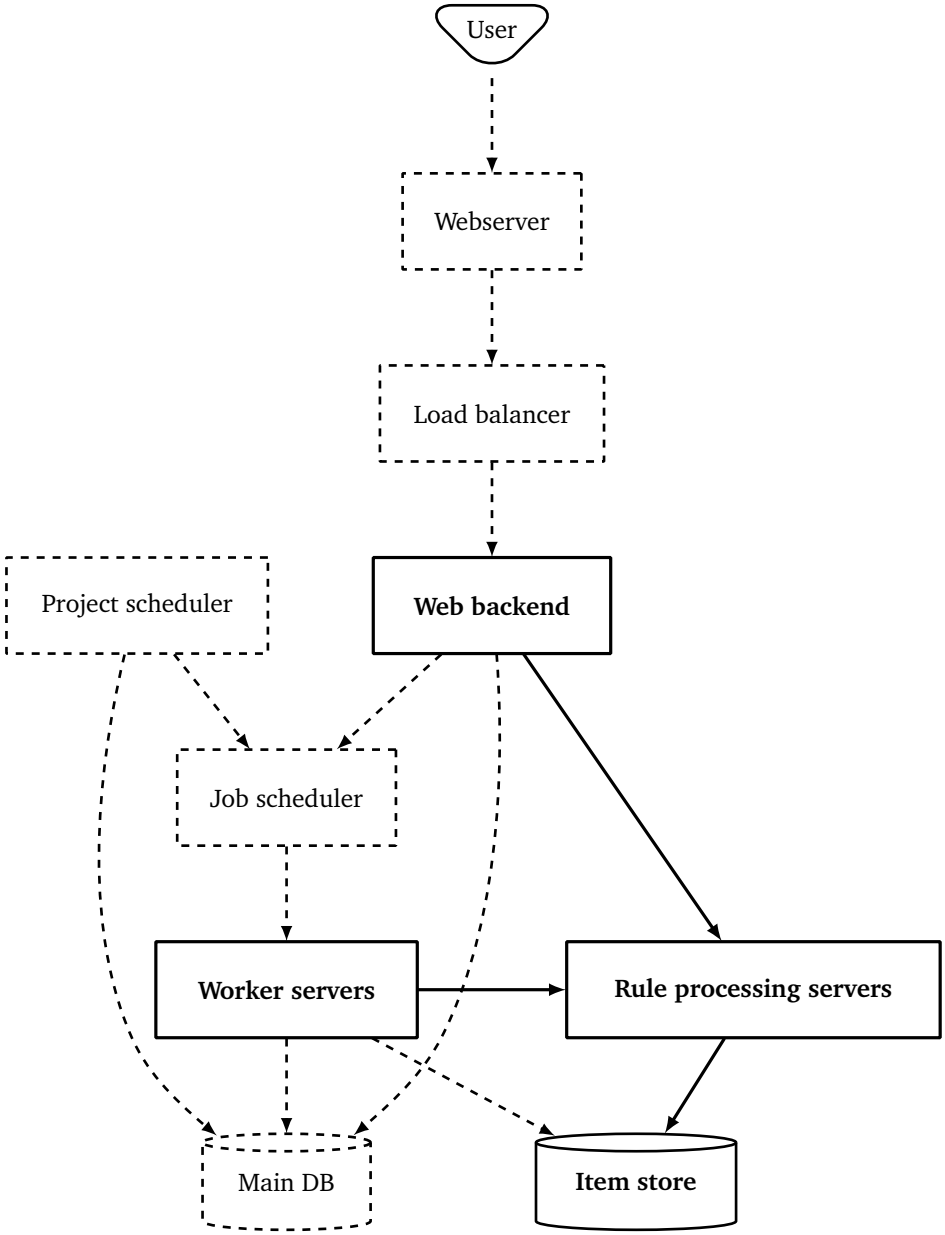

Figure 1: An overview of the components of the product feed processing system and how they communicate. Only the components in bold are relevant to this project's scheduling problem.

3 The rule processing system

In this section we describe the rule processing system, the main focus of this thesis.

The rule processing system is responsible for applying the configured data processing rules to the product data in the rule processing phase. It is implemented as a custom-built webserver using the Haskell programming language and exposes a HTTP API which accepts requests to execute *rule processing jobs*. During a rule processing job the rule processing engine will download the project's product data from the item store and apply the processing rules that were supplied to it.

The components of the product feed processing system that can submit rule processing jobs are the web backend and the worker servers. Rule processing jobs submitted by the web backend are always marked as high priority, for tasks submitted by the worker servers the priority depends on the priority of the job that the worker is running. In the context of a rule processing job the component that submitted the task is called the *submitter*.

3.1 Rule processing requests

The rule processing system accepts rule processing requests via HTTP. A rule processing request contains information needed for the rule processing system to run a single rule processing job.

Rule processing requests consist of the following data:

- Which project the rule processing job belongs to. This determines which set of product data should be loaded from the item store.
- The priority of the rule processing job. Either 'high' or 'low'.
- Which processing rules to apply, and in what order.

  Each rule processing request contains two groups of processing rules: the *shared rules* and the *channel-specific rules*. The shared rules are applied before the channel-specific rules, and as their name implies the shared-rules may be shared across multiple export channels.

  The split between shared and channel-specific rules can be used to speed up concurrent processing requests for projects with multiple export channels, by sharing intermediate results. This behaviour is described in more detail in section 3.4.
- Timeout information for the rule processing task. The timeout information specifies maximum timespans the task may remain in a certain state. If any of these timeouts is exceeded, the rule processing task is aborted and an error is returned to the submitter.
- A *session identifier*. The session identifier is stored in the logs and can be used to correlate requests from the same submitter that were retried across different rule processing servers.
- Whether the result of the rule processing job should be cached.
- Various options affecting the presentation of the requested operation's results, such as whether the results should be sorted or whether the results should be filtered according to a search term.

Example

To demonstrate the rule processing system we show a rule processing request for a fictional merchant selling T-shirts. The merchant's product feed is shown in table 1. The product feed contains five fields for each product: the product's identifier, title, color, stock level, and price. We assume that the

id	title	color	stock	price
2	Test shirt	white	23	
3	T-shirt “Triangle”	blue	34	15.00
4	T-shirt “Triangle”	red	0	15.00
5	T-shirt “Hexagon”		78	18.00

Table 1: Example product feed before processing.

id	title	color	stock	availability	price
3	T-shirt “Triangle” Blue	Blue	34	in stock	15.00
4	T-shirt “Triangle” Red	Red	0	out of stock	15.00
5	T-shirt “Hexagon”		78	in stock	18.00

Table 2: Example product feed after processing.

target platform accepts these fields as-is and requires one extra field ‘availability’ indicating whether a product is in stock or not, and that the target plaform requires advertised product names to be unique.

To process this product feed the merchant defines the following rules:

1. If a product has no valid price (price less than or equal to zero or missing entirely), remove it.
2. If a product has a stock level greater than zero, set its availability to ‘in stock’. Otherwise, set the availability to ‘out of stock’.
3. If a product has a color set, capitalize the color and append the color to the product’s name.

Listing 1 shows the representation of this ruleset as a rule processing request, with some presentation and debugging-related information omitted. The resulting product feed after applying this rule processing request to the merchant’s product feed is shown in table 2.

3.2 Prefetch requests

The rule processing system is mainly used to apply processing rules for the product feed processing system as described above. The rule processing system also accepts various other request types, of which one is relevant for this thesis project’s scheduling problem: the *prefetch request*.

A prefetch request is used to indicate to the rule processing system that it is likely that it will receive a high priority rule processing request for a given project in the near future. Prefetch requests are automatically submitted by the web backend when it is detected that a user of a particular project has opened the Channable application.

When a prefetch request for a project is received, the rule processing system will download the relevant project’s product data to its cache, so that any processing requests for that project can be completed more quickly. This means that after executing a prefetch request for a project the rule processing server will not yet have performed any actual work, but that subsequent requests for processing for this project will be able to run faster on that server.

3.3 Rule processing request handling

A rule processing request goes through the following phases after being received by a rule processing server:

```
{
    "session_id": "5531f48d-74f0-4b52-8203-470d5ed8aa90",
    "project_id": 12345,
    "priority": "high",
    "timeouts": {
        "queue_seconds": 15,
        "evaluate_seconds": 120,
        "stream_seconds": 300
    },
    "shared_operators": [
        {
            "name": "Exclude items without valid price",
            "condition": [
                {
                    "field": "price",
                    "filter": "is_empty",
                    "children": [
                        {
                            "field": "price",
                            "filter": "is_less_equal",
                            "value": {"type": "static", "value": 0}
                        }
                    ]
                }
            ],
            "actions": [ {"field": "all", "action": "remove"} ]
        }
    ],
    "operators": [
        {
            "name": "Set availability",
            "condition": [
                {"field": "stock", "filter": "is_greater_than", "value": {"type": "static", "value": 0}}
            ],
            "actions": [ {"field": "availability", "action": "set", "value": "in stock"} ],
            "else_actions": [ {"field": "availability", "action": "set", "value": "out of stock"} ]
        },
        {
            "name": "Add color to product title",
            "condition": [
                {"field": "color", "filter": "is_not_empty"}
            ],
            "actions": [
                {"field": "color", "action": "texttransform", "value": "capitalize"},
                {
                    "field": "title",
                    "action": "combine",
                    "value": [
                        {"type": "field", "value": "title"},
                        {"type": "static", "value": " "},
                        {"type": "field", "value": "color"}
                    ]
                }
            ]
        }
    ]
}
```

Listing 1: Example rule processing request as submitted to a rule processing server.

1. *Parsing*: the rule processing request is parsed into an abstract syntax tree.
2. *Compilation*: the rule processing request is compiled into an intermediate representation.

   During the compilation phase the rule processing job is broken up into multiple smaller *joblets*: each of these joblets represent a part of the requested rule processing task. The division into joblets is described in more detail in section 3.4.
3. *Optimization*: the compiled rule processing task is processed to remove steps that don't affect the output of the task. Examples of such steps are rules that affect the value of attributes that the submitter requested to be omitted or rules with conditions that can be statically determined to be unsatisfiable.
4. *Queueing*: the joblets of which the rule processing task consists are submitted as a group to the executor's job queue.
5. *Evaluation*: the *executor* becomes responsible for executing the rule processing job. The behaviour of the executor is described in more detail in section 3.4.

The submitter will keep its connection to the rule processing server open until it receives the result of the rule processing task. If the submitter closes its connection, the executor is told that the joblets of the submitted rule processing task can be canceled.

3.4 The executor

The *executor* is the subsystem of the rule processing engine that performs the actual rule processing tasks. The executor manages the CPU cores of the rule processing server and schedules the joblets it receives on those cores according to its scheduling logic.

Each rule processing job is split into multiple smaller joblets before submission to the executor, as described in the previous section. Various exceptions exist, but most rule processing jobs are split into joblets of the following types:

- *FeedSource*, which fetches the product data to process from the item store,
- *SharedOperators*, which applies the shared processing rules,
- *Operators*, which applies the channel-specific processing rules, and
- *OutputFeed*, which converts the processed product data to the requested output format and returns it to the submitter.

Prefetch requests only result in a FeedSource joblet.

Each of these joblets depends on the result of the previous joblets in the list. Between the execution of each joblet the product data is stored in the rule processing server's cache. This process is described in more detail in section 3.5.

The executor maintains a priority queue of joblets which are to be executed: joblets are ordered first by priority, then by submission time. When the executor determines capacity is available, it will pop the highest-priority joblet from its queue, determine how many CPU cores the joblet can use and begin executing the joblet with this number of cores. Once a joblet is being executed it will not be interrupted before it is done, unless the rule processing server encounters an error or is shut down.

The division of rule processing jobs into joblets allows the executor to still pause the execution of a job: if a low-priority rule processing job is being executed and a high-priority rule processing job

is received, the executor may pause the low-priority job by delaying execution of the joblets for the low-priority job until the high-priority job has completed.

The executor logs debugging information on each executed joblet to the item store database. This information includes information such as the joblet's evaluation time, queue time and the amount of memory used by the joblet. This information could theoretically be used to assess the rule processing system's overall performance, but this information is unfortunately slow to query, so it is unsuitable for making scheduling decisions at runtime.

The executor also exposes metrics on the state of the executor to the monitoring system, which can be queried efficiently at run-time. The exposed information includes aggregated statistics on the performance of recently executed joblets and the state of the joblet queue. The most interesting of these metrics is the 'stalled promises' metric: this metric indicates the amount of joblets that the executor has determined to be available for execution, but could not yet actually execute because of a lack of CPU resources. This metric can be used to determine a server's load level more granularly than the binary 'healthy'/'unhealthy' information provided by the service discovery system.

3.5 Caching

The executor stores the result of each joblet it executes in the rule processing service's cache. This serves two purposes: it is required to pass data between the job's joblets due to the rule processing service's design, but also allows the reuse of earlier computation if the preceding data did not change. This is possible because (like the whole rule processing jobs) the joblets are also pure: given the same input a joblet will return the same output.

The executor uses this fact to deduplicate joblets that compute the same data: if a rule processing job is started which depends on data that is already in the cache or that is already scheduled to be computed, the executor will reuse this result instead of scheduling a duplicate joblet to compute the same data. This deduplication mechanism can speed up subsequent rule processing jobs for the same project if they run on the same server.

Figure 2 shows an example of this de-duplication: in this example the result of submitting three rule processing jobs is shown. In this example all three jobs specify the same set of product data as their source and the second and third job specify the same set of shared rules. By the use of deduplication the executor can avoid executing two FeedSource joblets and one SharedOperators joblet, speeding up the rule processing jobs submitted after the first rule processing job.

The cache is *ephemeral*: it is cleared when a new version of the rule processing engine is deployed, or when the server where the rule processing engine is deployed restarts. This also happens when a rule processing server is pre-empted as described in section 3.6: this implies that any data on pre-emptible rule processing servers remains in cache for at most 24 hours.

When new data is to be stored in the rule processing server's cache, but the cache is full, the rule processing server will evict the oldest entries from the cache, until the server has enough space to store the new data.

The rule processing servers expose information on which projects are stored in the servers' caches via a HTTP API, which allows querying this information from another server. This information is currently only used for debugging purposes.

3.6 Pre-emptible infrastructure

The rule processing system is partially deployed on *pre-emptible servers*. This means that a number of the servers on which the rule processing system runs are backed by spare capacity of the hosting provider, which reduces cost.

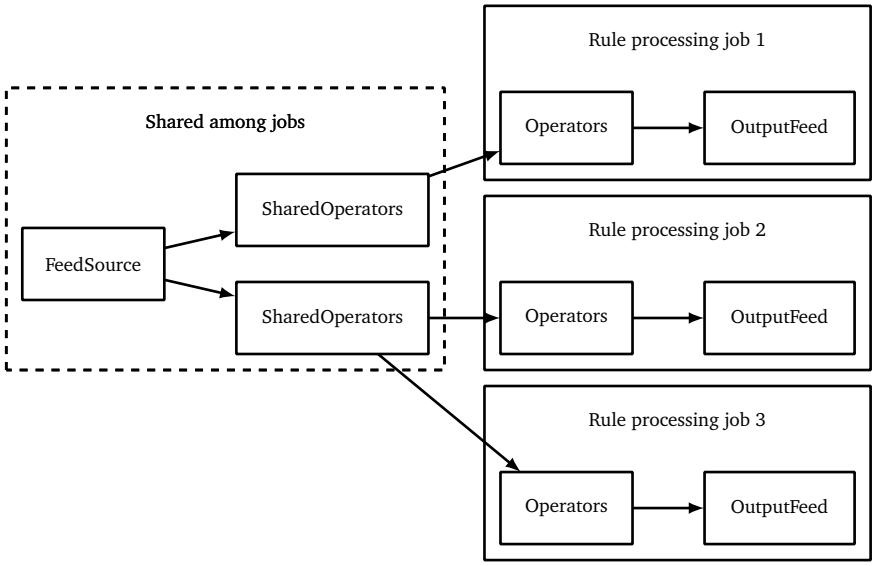

Figure 2: Example of joblet deduplication for three rule processing jobs of the same project, with the same set of product data and only two distinct sets of shared rules.

The trade-off of using pre-emptible servers is that the hosting provider may shut down the servers at any moment (with 30 seconds of advance notice) if the capacity is needed for non-preemptible servers. The hosting provider will also terminate pre-emptible servers after they have run for 24 hours, regardless of whether the server's capacity is needed elsewhere.

To be able to deal with servers being shut down on relatively short notice the rule processing service has been designed to not contain any local state other than the cache. This allows the rule processing servers to shut down at any moment, with the only downside of this shutdown being that any rule processing tasks that are in progress are canceled.

When a rule processing server is pre-empted it is removed from the service discovery system, so that the web backend and worker servers will no longer attempt to submit rule processing jobs to it.

A monitoring component in the product feed processing system checks every five minutes if any servers have been pre-empted. If any pre-empted servers are found, the system issues a command to restart the servers, after which they automatically rejoin the service discovery system and can receive rule processing requests again.

4 The scheduling problem

When a submitter (either the web backend or a worker server) needs the results of a rule processing task, it needs to select a rule processing server to execute the rule processing task. In this section we describe the currently-implemented method for a submitter to select a server and the advantages and shortcomings of this method. We also describe the relevant available data which could be used to make a better scheduling decision, and the characteristics of the described scheduling problem.

4.1 Current method

The currently-implemented method to select a rule processing server is a pseudo-random selection of servers based on the project identifier and the set of rule processing servers that are currently available.

The selection process works as follows:

1. Query the service discovery system to retrieve the list of available rule processing servers.
2. Order this list of servers alphabetically.
3. Create a pseudo-random number generator using the project's identifier as a seed.
4. Randomize the order of the retrieved servers using this pseudo-random number generator.
5. Attempt to submit the rule processing task to the first three servers in the shuffled list of available servers. Fall back to the next server in the list if an earlier rule processing server fails to process the rule processing task.
6. If any of the three candidate rule processing servers produce a result, exit successfully. Otherwise, give up.

4.1.1 Advantages

The described host selection process has various advantages:

- The selection process is "stable": given the same set of available rule processing servers and project ID the system will select the same rule processing servers to submit the task to.

  This is a design feature: the selection process was originally intended to divide the workload evenly between the various rule processing servers, but still attempt to process the tasks for a single project using the same server. Using the same server for the same project is advantageous because this allows the server to reuse the cached product data, reducing the duration of jobs since the input data doesn't need to be re-downloaded from the item store.

- Because the selection process is executed on the submitter, the selection of rule processing hosts is not subject to a 'single point of failure'.
- The described selection process is relatively easy to implement.

4.1.2 Disadvantages

During the time the described selection process has been in use at Channable the rule processing system was identified to not work well in practice.

Various disadvantages of the selection process were identified:

- The selection process does not take the rule processing servers' current workload into account. The selection process only checks whether rule processing servers are 'healthy' in the service discovery system, i.e. online and not crashed, but not whether the servers are actually able to accept new jobs.

  It can happen that a server is selected which is already at its processing capacity, which results in the job having to wait in the server's queue. This can happen even if other servers are idle, which is not ideal.

- The rule processing workload is not actually evenly distributed among the rule processing servers because, the size of the various projects for which rule processing tasks are executed varies massively.

  The workload induced by a single project can vary because of several factors which vary between projects:

  - The size of the product data: some projects contain hundreds of items, while other projects process millions of items.
  - The schedule of the project: some projects are configured to run their jobs only once a day, while other projects are configured to run hourly or even every 15 minutes.
  - The number of export channels: some projects have only a single export channel, while other projects have hundreds.

  Because these factors exhibit 'long-tail' behaviour the impact of assigning a project to a rule processing server varies massively, with most projects relatively close to each other but some outlier projects having a disproportionate effect on the servers they're scheduled on.

  This imbalance leads to some rule processing servers often being overloaded, while other rule processing servers are idle at the same time.

- The rule processing server selection process is not robust against changes in the set of available servers: if the set of available servers changes, the assignment of almost all projects to rule processing servers changes as well. The exact number of projects that get reassigned to a different server when a new server comes online or goes offline would ideally be small, so that the servers' caches remain useful. In practice the currently-used server selection process reassigns more than half of all projects to a different server each time a server is added or removed.

  Server additions and removals also happen more often than when the selection method was designed: when the described selection process was implemented, Channable did not yet use preemptible infrastructure (see section 3.6), so servers went on-/offline less often.

- Lastly, the server selection process was designed to ensure that projects get scheduled on servers where the project is in cache. However, this is not actually checked: if a project is in cache on some other server than the server selected by the selection process, the rule processing job will still run on the server picked by the selection process, even though it might run more quickly on the server where the project is in cache.

It is clear that a different host selection process is needed.

4.2 Available data

The current rule processing host selection process uses only the set of available servers and the project identifier to select a rule processing server.

However, there is more information available which could be used to make a better scheduling decision:

- The information in the rule processing request as detailed in section 3.1,
- Information on the size of the project's data in the item store,
- Information on historical performance of rule processing jobs for the same project,
- Information on the load of the rule processing servers: how many rule processing jobs it is executing and how many rule processing tasks are in the server's backlog,
- Information on which project's data is in cache on each rule processing server as described in section 3.5.

4.3 Scheduling factors

A new host selection process should take the following factors into account:

- The selected server must be available and able to accept new job requests. This information is available in the service discovery system.
- The selected server ideally has the project's data in its cache, so it does not need to be downloaded before the job can start. This information can be retrieved from each rule processing server.
- The selected server ideally has no backlog for the job's priority, so the rule processing job can start immediately. This information can be retrieved from the monitoring system.

  If the server has the project's data in cache, but the expected time the job will have to remain queued is longer than the time it takes to download the project's data, it may be better to select a different server, as waiting for the server where the data is in cache won't actually provide a speed benefit.
- Servers should ideally not be used if they are preemptible and it is expected that the server will shut down while the job is executing (section 3.6), to prevent having to reschedule the aborted processing jobs on another server.

4.4 Scheduling characteristics

The described scheduling problem of assigning rule processing jobs to rule processing servers matches several common properties in scheduling problems.

Firstly, the described problem is an *online* scheduling problem: rule processing tasks may arrive at any time and a scheduling decision needs to be made immediately, though that scheduling decision may be "retry at a later time".

Secondly, the scheduler needs to be *non-clairvoyant*: it is not known how long the rule processing tasks will take to execute. The time needed to execute rule processing jobs can still be measured, which might allow prediction the duration of a rule processing task based on historical data. However, currently no such prediction system is implemented.

Third, the scheduler cannot currently be *preemptive*: after a job has been assigned to a server the scheduler cannot decide to pause or terminate the job in favor of another higher-priority job, because the rule processing system does not currently support this.

Lastly, the scheduler needs to be *fault-tolerant*: rule processing jobs may fail and the rule processing servers may become unavailable at any time. If this happens scheduling still needs to continue even though the rule processing system may be in a degraded state.

5 Related work

In this section we describe found literature that is relevant for solving our scheduling problem.

5.1 Prior project

Paweł Ulita performed an earlier master's thesis project [11] in 2020 on the same product feed processing system described in section 2. Ulita described the system and performed experiments on a partial replica of the rule processing system, to predict how implementing a centralised scheduler with various scheduling policies would affect the production system's performance.

In his experiments Ulita compared the currently-used scheduling policy (as described in section 4.1) with a scheduling policy based on a 'caching plan', a pre-computed static assignment of projects to rule processing servers. To perform this comparison Ulita captured a workload trace containing the rule processing jobs that were submitted over a 24-hour period and replayed these jobs on identical virtual machines. Ulita noted that the method he used to create a replica of the data in the item store resulted in an incomplete copy, which resulted in 25% of the jobs recorded in the workload trace being unable to run.

Ulita's project defined three optimisation objectives: increasing the system responsiveness for high priority jobs, increasing punctuality for low priority jobs, and improving the cache hit ratio for the rule processing servers.

For the first two optimisation goals Ulita evaluated the performance of his scheduler by reporting on two metrics: the waiting time and *slowdown* experienced by the jobs. The waiting time of a job was defined as the time between a job being received on a rule processing server and it starting execution and the *slowdown* was defined as $\frac{W}{W+E}$, where W is the job's waiting time and E is the job's execution time.

For the third optimisation goal Ulita reported on the number of cache evictions on each rule processing server, because the cache hit ratio itself was not possible to record.

Ulita showed that his proposed scheduler reduced the mean waiting time and mean slowdown for both the low priority and high priority rule processing jobs, when compared to the baseline experiment. However, the variance in waiting times per rule processing server in the experiment was still relatively high and the variance in slowdown per rule processing server actually increased for high priority jobs. Ulita also showed that his proposed scheduler increased the amount of cache evictions across all rule processing servers, suggesting a decrease in the cache hit ratio.

5.2 Schedulers

Various schedulers have been proposed which attempt to address similar scheduling problems to the scheduling problem described in this document.

Zaharia et al. [12] describe *HFS*, a scheduler for the Hadoop cluster computing system in use at Facebook. One of the design goals of HFS is data locality, i.e. ensuring that tasks are scheduled on nodes where the input data is immediately available or more easily loaded than on other nodes. The authors show that it is often beneficial to wait a short amount of time for nodes where data is available to become available, instead of scheduling the tasks on other nodes (where data first has to be loaded). This approach improved the response times of small CPU-bound jobs by 5× and doubled the throughput for an IO-heavy workload trace.

Ananthanarayanan et al. [1] describe *PACMan*, a caching system that speeds up data processing jobs by ensuring that the jobs' input data is available in a worker-local cache. The authors show that for their workload, which is based on processing jobs at Bing and Facebook, the input sizes and task counts per job are *heavy-tailed*. This means that the workload mainly consists of jobs with relatively

small input sizes which run relatively infrequently, with a small number of significant outliers in input size or run frequency. The authors evaluate their caching system by running experiments based on workload traces from Bing and Facebook and report that their caching system significantly reduced the completion times of jobs.

Boutin et al. [2] describe *Apollo*, a distributed scheduler in use at Microsoft. Among the techniques used in Apollo is estimation-based scheduling, which uses estimates of the time it would take to load a job's input data to a node, as well as the expected job queue time at a node. The authors evaluate Apollo by comparing system metrics before and after the scheduler was deployed. The authors show that their scheduler reduces the variance in wait time across hosts even without using estimation-based scheduling, and that enabling estimation-based scheduling further increases the quality of the scheduling decisions.

Garefalakis, Karanasos, and Pietzuch [6] describe *Neptune*, a centralised preemptive scheduler for the Apache Spark system in use at Microsoft. Neptune supports running both low-priority batch jobs and high-priority streaming jobs on the same executors, suspending low-priority jobs in favour of high-priority jobs if necessary. The authors evaluate Neptune's performance by running experiments based on workload traces from various sources and comparing the scheduler's behaviour with the default schedulers in Apache Spark and several alternative policies. The authors show that their scheduler can maintain high throughput of low-priority jobs while also reducing the request latency for the high-priority streaming jobs.

5.3 Hedged requests

Dean and Barroso [5] describe techniques which Google uses to reduce variance in request latencies for data processing systems. Among these techniques are *hedged requests*, the practice of sending requests to multiple servers concurrently and then using whichever response is delivered first.

By executing requests on multiple servers concurrently the system's load is increased, but the worst-case request latency can be reduced significantly, since the chance that a request takes a long time to process is decreased. The authors also describe how delaying the second request a short amount of time can find a trade-off between worst-case latency reduction and the extra load induced by the duplicated requests.

Primorac, Argyraki, and Bugnion [8] describe *Lædge*, a hedging policy which only applies request hedging when the system load is low to not hinder the system's throughput when under high load. The authors show that this hedging policy outperforms 'naïve' hedging if the probability of requests experiencing hiccups is relatively low.

5.4 Load shedding

Various approaches exist to detect and mitigate overload in request-processing systems. One such approach is *load-shedding* or *circuit breaking*, which is the practice of dropping requests when the system is detected to be overloaded or returning errors. Traditional approaches to circuit breaking only allow operators to drop requests based on static thresholds like a certain number of queued requests.

Sedghpour, Klein, and Tordsson [9] describe a dynamic circuit breaker that allows the threshold at which a circuit breaker activates to vary depending on the average response time of the system. The authors show that this method outperforms static circuit breaking in an experiment, with more requests being able to complete successfully at the same system load.

Zhou et al. [13] describe *DAGOR*, a load-shedding method for microservices in use at WeChat. DAGOR assigns each request two priorities: a 'business priority' corresponding to the type of request and a 'user priority' corresponding to the user for which the request is to be executed. DAGOR uses

the average waiting time of requests in a server's queue to determine whether a server is overloaded. When a server is determined to be overloaded, DAGOR first sheds load by making the server drop less-important requests, i.e. requests with a low business priority and low user priority. This approach ensures that more important requests have a higher chance to succeed if the system is under moderate load. By randomizing the user priority over time the authors ensure better fairness, by ensuring that the system varies which users' requests get dropped first when the system is at load. The authors evaluate DAGOR by performing experiments to compare DAGOR to other overload control systems and show that DAGOR is able to outperform these systems, by allowing a higher amount of requests to succeed.

Cho et al. [4] describe *Breakwater*, a load-shedding method for microservices targeting a μs response time. Breakwater controls the system load by issuing 'service credits' to clients and requiring clients to have a valid service credit before being allowed to submit a request. By varying the amount of service credits issued based on the system load the incoming request rate can then be controlled. The authors show using experiments that Breakwater is able to recover from a sudden load spike quicker than other overload control systems like DAGOR.

Part II
Our project

6 Project outline and goals

In this section we describe the project's research question and optimization goals.

6.1 Approach

Our project's research question as defined in section 1 is: *How to dynamically schedule rule processing jobs to evenly divide the workload among a number of worker servers?*.

To answer our research question we follow a similar approach to Ulita [11]. This approach consists of running experiments on a cluster of servers, using captured rule processing jobs from the production environment.

By repeating the experiment multiple times with different scheduling policies we can compare the effectiveness of a certain scheduling policy.

In section 7 we describe the setup of our experiment environment in detail. In section 9 we describe the scheduling policies we evaluate in detail.

6.2 Optimization goals

We focus on two optimization goals for our project:

1. **Reducing the duration of rule processing jobs:** we want to improve the performance of the rule processing servers by having requests sent to them complete sooner.

   To quantify this improvement we will compare the durations of the rule processing jobs.

2. **Reducing the waiting time of rule processing jobs:** we want the majority of a rule processing job's duration to be spent doing useful work, rather than waiting for capacity to be available on the rule processing server.

   To quantify this we will focus on reducing the absolute waiting time of the rule processing jobs (i.e. the time a job has to wait in a rule processing server's queue). We will also compare the *slowdown* as defined by Ulita [11] (section 5.1).

6.3 Scope

The rule processing system and other components of the product feed processing system are under active development separate from this project. To limit the scope of the project and to ensure the results are still applicable to the production system once the project concludes we only investigate the effect of changing the method for selecting rule processing servers on the overall system's performance.

We realise that it might also be possible to improve the rule processing system's performance by optimizing the performance of the system itself, but consider this out of scope for this thesis project: this is already being investigated by Channable's infrastructure team.

7 The experiment environment

In order to run our experiments we need an experiment environment, in which we can reproducibly run a rule processing workload and observe the outcome. In this section we describe the setup of the experiment environment we used for our project.

7.1 Design

The design for the experiment environment used for our experiments is based on the design by Ulita [11], with some additions to update the experiment environment to the changes that have been made to the production environment since Ulita's project concluded.

Our experiment environment consists of 17 virtual machines, divided in four 'classes':

- Fourteen rule processing servers, with a configuration which matches the production environment as closely as possible,
- One 'submitter' server, responsible for generating a workload for the rule processing servers by replaying traffic captured in the production environment,
- One monitoring server, responsible for running the monitoring system, service discovery system and scheduler, and
- One database server, responsible for storing a copy of the production environment's item store database.

  Like the rule processing servers this server's configuration must match the configuration of its counterpart in the production environment as closely as possible, in order for the performance in the experiment environment to be representative of the production environment.

Next to the servers our experiment environment also includes a Google Cloud Storage bucket, which is used to store a copy of the item store objects.

An overview of the experiment environment's design is shown in figure 3. Like in Ulita's design the following changes are made compared to the production environment described in section 2:

1. A submitter component is introduced, which is responsible for submitting the rule processing jobs to the rule processing servers. The submitter replaces the web backend and worker servers.

   The submitter is provided with a file containing a 'workload trace', which defines which requests the submitter should send at what times.

2. A scheduler component is introduced, which is queried by the submitter to determine which rule processing server a job should be scheduled on. Like in Ulita's design, the scheduler does not actually schedule the job, but only provides a recommendation to the submitter on which server to use.

   The scheduler component is described in more detail in section 7.3.

New (compared to Ulita's design) is that:

1. Our experiment environment also includes a Google Cloud Storage bucket to store a copy of the item store objects. This change is necessary because the production environment also started using a Google Cloud Storage bucket after the conclusion of Ulita's project.
2. Our scheduler component also queries the monitoring system for information needed in the scheduling decision, where Ulita's scheduler only relied on the service discovery system and rule processing servers.

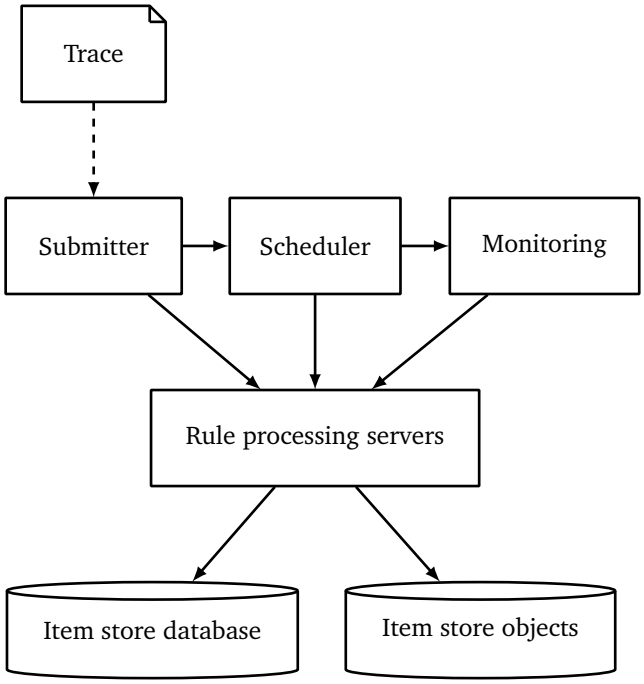

Figure 3: An overview of the components of our experiment environment and how they communicate.

7.2 Submitter

The submitter is the component responsible for replicating the workload of the production environment in our experiment environment. The submitter is provided with a workload trace file containing a set of captured rule processing requests along with the 'offset' of the request, i.e. the time relative to the start of the experiment when the request should be sent. The offset of a request is used to determine the time the request should be sent, by adding the offset to the start time of the experiment. To run an experiment the submitter iterates over the requests in the workload trace file. The submitter assigns each request to a worker thread, which will perform the following steps:

- Wait until the request's send time, i.e. wait until the current time is equal to the start time of the experiment plus the request's offset. If this time has already passed, continue immediately.
- Send a scheduling request to the scheduler to determine to which server the rule processing request should be sent. The scheduling request only includes the request's priority ('low' or 'high') and which project the request belongs to.

  If the scheduler is unreachable or reports that no servers are available, retry later using an exponential backoff strategy [3].
- Send the rule processing request to the rule processing server indicated by the scheduler and wait for the rule processing server's response.

  If the rule processing job succeeded, mark the request as succeeded in the submitter's log. If the rule processing server reports an error or is unreachable, mark the request as failed and record the error type in the submitter's log, and do not retry.

Not retrying failed rule processing jobs is a conscious decision: the submitters in the production environment already perform retries when a rule processing job fails unexpectedly and each retry is already present in the workload trace file, because the workload trace file includes all requests to the rule processing servers. Performing another round of retries in the submitter would increase the load on the experiment environment when compared to the production environment or cause inconsistencies between experiment runs if a particularly computationally expensive request is retried during one experiment but succeeds right away during another.

By providing the submitter with the same workload trace file for all of our experiments we can ensure that the workload for each of our experiments is the same.

7.3 Scheduler

The scheduler is the component responsible for assigning rule processing jobs to rule processing servers. The scheduler accepts *scheduling requests* over HTTP, containing a subset of the information contained in a rule processing request. For each received scheduling request, the scheduler determines which rule processing server the job should be run on and returns this selection to the client.

The scheduler uses three external information sources to make its scheduling decision:

- The service discovery system, to determine which rule processing servers are able to accept requests,
- The monitoring system, to retrieve relevant statistics on the rule processing servers that are needed for the scheduling decision, and
- The rule processing servers themselves, to determine the state of each rule processing server's cache.

Because these information sources can be slow to respond or intermittently available the scheduler queries these sources asynchronously every five seconds and caches the responses, so that when a scheduling decision has to be made the information is available already. This approach allows the scheduler to make faster scheduling decisions, at the cost of always operating on slightly-outdated information.

The scheduling behaviour of the scheduler is configurable: the scheduler uses a separately-configurable *overload detection policy* to determine which servers are eligible to run rule processing jobs, as well as a *host selection policy* to determine which of the eligible hosts to schedule a job on. In section 9 we describe the overload detection policies and host selection policies implemented in our scheduler.

7.4 Notable differences between the production and experiment environments

The environment used to run the experiments for this project was set up using a setup process based on Channable's existing Infrastructure-as-Code (IaC) tooling, in order to create an environment that mimics the production environment as closely as possible.

The reuse of the existing IaC tooling means that the production and experiment environments both run the same versions of the operating system, supporting software like the service discovery system and the same version of the rule processing engine as were in use at the time the workload trace was captured.

In this section we describe the known differences between the production and experiment environments, along with the expected impact of these differences on the performance of the experiment environment as compared to the production environment.

7.4.1 Static dataset

As described in section 2.4 the rule processing engine operates on input data stored in the item store. In the production environment each project's data in the item store is updated at least once every 24 hours, with many projects being updated more often, but in the experiment environment the project data is a snapshot and never updated.

Due to this difference we expect that in the experiment environment less fetches of the input data might be needed: because the input data does not change while the experiment is ongoing jobs are more likely to be able to reuse job data already cached on a rule processing server.

7.4.2 Slower disks

In Channable's production environment all servers are equipped with SSD-backed storage drives, which support high performance in terms of supported I/O operations per second (IOPS) and sustained throughput.

We decided not to use SSD-backed drives in the experiment environment due to cost concerns. Instead, the database server was set up to use 'balanced' drives, which offer a trade-off between the performance of a SSD-backed drive and the lower cost of a HDD-backed drive. Balanced drives are 1.25 times slower than SSD-backed drives, supporting a similar amount of IOPS but a lower sustained throughput level.

All other servers were set up using 'standard' drives, which are backed by HDDs. These drives are about 13 times slower than SSD-backed drives. The non-persistent cache storage on the rule processing servers was set up using the same type of drive as used in Channable's production environment.

Because we used a slower drive type for the item store database server we expect that accessing information in the item store database will be slower than in Channable's production environment, as the data cannot be read as fast from the database server's disks. We expect this difference to only have a slight impact on the performance of the experiment environment: the majority of the item data is stored in Google Cloud Storage, which is not affected by this difference in drive type.

Because of the slower drive type used for the rule processing server we expect that the rule processing servers will be slower to start, but that runtime performance will not be affected. When a rule processing server has started, only its volatile cache storage is used for performance-sensitive operations, and the cache storage in our experiment environment does use the same drive type as the production environment.

7.4.3 Different timeout, retry behaviour on rule processing requests

In Channable's production environment requests to the rule processing servers are subject to retries and timeouts. If a request fails due to a problem that is likely to be transient, the request may be automatically retried after waiting a short amount of time, using the exponential backoff with jitter strategy [3].

In the production environment requests are subject to two different times of timeout: the connection timeout and the read timeout. The connection timeout limits the time the client will wait until the rule processing server acknowledges the request. The read timeout limits the time the client will wait for the server to start sending the response to the client's request. If a request fails due to the connection timeout, the next request's connection timeout is doubled.

The HTTP library we used to build the submitter described in section 7.2 does not support differentiating between a connection timeout and read timeout, and only exposes a 'total' timeout parameter. For this reason it was not easily possible for us to set separate connection and read timeouts in the experiment environment like in the production environment.

To determine whether this would be a problem we inspected the production environment's logs. These logs showed that 98.15% of the requests to the rule processing servers within the capture period were able to connect to a rule processing server on the first try, while only 0.11% of the requests failed due to the connection timeout.

Based on this rate we decided that removing the connection timeout was acceptable and did not implement an alternative for it in the submitter. Requests in the experiment environment are still subject to a timeout: we set the total timeout for a request in the experiment environment to the sum of the connection timeout and read timeout in the production environment. As described in section 7.2 we also purposefully did not implement a retry system, as the captured requests in the workload trace are already the result of a retry process.

We expect that this change could cause our experiment environment to perform worse than the production environment, if requests are scheduled on an overloaded server. If a server is overloaded it becomes unable to accept new connections, which can be detected in the production environment by the connection timeout being exceeded. Because there is no separate connection timeout in the experiment environment the submitter will have to wait for the duration of the full request timeout, which is considerably longer, so the submitter will spend more time waiting for the request to time out. This increase in request handling time should not negatively affect the performance of our submitter as requests are handled asynchronously, so the submitter should be able to perform useful work in the meantime by sending other requests.

7.4.4 Different pre-emption behaviour

In Channable's production environment, 10 out of the 14 rule processing servers are preemptible servers, as described in section 3.6. Due to cost concerns we decided to set up all 14 rule processing servers in the experiment environment as preemptible servers.

The effect of this change is that, unlike the production environment, all rule processing servers can be shut down arbitrarily by the cloud provider if capacity is needed, which could theoretically cause the rule processing system to become unavailable. In Channable's production environment this has never happened, so we expected this change to have no impact on the experiment environment's performance.

We unfortunately discovered that this change did have another unforeseen effect on the performance of the experiment environment. This effect is described in section 10.1.

7.4.5 Empty caches on experiment start

When an experiment is started the database and all rule processing servers are (re)started, to ensure that they're in a known starting state. This is done to ensure that there is no data in the servers' caches: if a server would have cached data it might affect the runtime of jobs scheduled on that server by making jobs appear to run faster than they really are.

The expected effect of this difference is that some jobs executed at the beginning of the workload trace run slower in the experiment than they did when executed in the production environment, where their input data or result might have been cached already. We expect the impact of this difference to be minimal, so we did not implement any measures to resolve this difference.

7.4.6 Different server selection in baseline selection method

The baseline server selection method makes use of the pseudo-random number generator (PRNG) in the Python standard library. Python's default PRNG implementation is based on the Mersenne Twister algorithm [7]. Because the scheduler is implemented in Haskell instead of Python it uses a different PRNG algorithm, using the SplitMix algorithm [10].

The effect of using a different PRNG algorithm is that the baseline host selection method will differ from the host selection method in production: given the same set of servers, the baseline host selection in the experiment will likely select a different server for a job than would be selected in the production environment.

As described in section 4.1.2 the number of requests that a server handles is not directly proportional to the actual computational load imposed on that server, so we expect that this difference will have only a small impact on the performance of the baseline experiment as compared to the production environment.

7.4.7 Request dependencies not preserved

Around 15% the rule processing jobs generated by the web backend are dependent on the result of an earlier job: the web backend will first submit a rule processing jobs with shared parameters and inspect the result of the job, then submit a second rule processing job containing some of the results from the first job as parameters. In the production environment these dependent jobs are guaranteed to not run at the same time, because the second job can only be submitted once the first job has finished.

The dependency between requests is not stored in the request data itself. Because of this the separation between requests cannot be guaranteed in the experiment environment: if the first request takes longer to execute than it did in the production environment, the second request may be sent while the first request is still processing. The expected effect of this difference is that the second request will take longer to execute than it did in the production environment, because it still has to wait for the first request to complete.

Because the information on the request dependencies is not available without modifying the web backend, no action was taken for this issue.

8 Capture and analysis of a new workload trace

In order to run experiments in the experiment environment described in section 7 we need a set of rule processing jobs to execute, as well as supporting datasets to ensure that the rule processing servers can actually execute the rule processing jobs. We refer to this combined data as 'the workload trace'.

In this section we describe the process we used for capturing the workload trace used in this project's experiments, as well as the corrections we applied to the captured data to make it suitable for use in the experiment environment.

8.1 Capture processes

The workload trace consists of three distinct datasets:

1. The request data, containing the HTTP requests sent to the rule processing servers by the web backend and worker servers,
2. The item store database, containing a copy of the item data stored in the production environment's PostgreSQL database, and
3. The item store objects, containing copies of the item data stored in the production environment's Google Cloud Storage bucket.

Each of these datasets are stored in different systems and require different approaches to be copied.

8.1.1 Request data

We captured the request data by using the 'request dump' configuration option implemented by Ulita in his earlier project [11]. This option, when enabled, causes the submitters (web backend and worker servers) to store a copy of each request sent to the rule processing servers on the submitter's disk.

We enabled the request dump option for a capture period of around 26 hours, after which we collected all captured requests from the various submitters on a central server. The resulting dataset contains 3,064,002 captured requests.

Due to how Channable's software deployments work configuration changes do not take effect right away and some time is needed to update the configuration on each of the submitters. This meant that the period during which requests were captured slightly varied for each submitter. To obtain a 24-hour request log of all submitters we determined when the request dump option had taken effect on all submitters, and filtered the captured request data to only the requests sent between this moment and 24 hours afterwards. This resulted in a dataset of 2,879,676 requests.

8.1.2 Item store database

We created a copy of the item store database by setting up a new PostgreSQL server and connecting the server to the production item store database using PostgreSQL's 'streaming replication' functionality. The streaming replication functionality allows a PostgreSQL server to receive a full copy of the data stored on another PostgreSQL server, along with the changes that are made to the original database after the initial copy of the data.

Using streaming replication solved the problems experienced by Ulita in his earlier project: Ulita used a series of 'batched copies' to copy subsets of the item store database, which temporarily interrupted production traffic to the item store database. A major downside of this approach was that the data in the item store database could (and did) change in-between copy operations, which caused 25% of the jobs in Ulita's workload trace to fail to run in his experiment environment.

By using streaming replication we were able to make an internally-consistent copy of the item store database, which allowed a higher percentage of jobs to run successfully.

8.1.3 Item store objects

The item store objects have different access characteristics than the item store database, which meant that copying the item store objects was relatively simple. To create a copy of the item store objects, we wrote a script that ran during the 26-hour request capture period to periodically copied all files from the production Google Cloud Storage environment into an isolated environment.

The use of an isolated Google Cloud Storage environment was mainly needed to prevent the data being deleted: the production environment is configured to automatically delete data from the item store objects after it has not been updated for 30 days, which is not desirable for the experiment environment.

8.2 Data correction steps

To determine whether the captured workload trace could be used to replay production activity in the experiment environment, we first ran tests using only the requests that were captured during the first ten minutes of the capture period.

During these tests we discovered several problems with the captured request data, which prevented the rule processing jobs associated with this data from being run. This section describes the major problems that were found and the steps we took to resolve these problems.

8.2.1 Missing product data in item store database

The product feed processing system periodically deletes old versions of the imported item data, as it normally only operates on the newest version of the product data.

We discovered that during the capture process some of the item data required by the captured requests had already been deleted from the item store database due to newer versions of the data becoming available, while we had expected this data to still be available at the time the capture period ended. This caused the jobs associated with the deleted item data to be unable to run, as they referred to now-deleted versions of the product data. For most of the affected jobs a later version of the product data was still available in the captured copy of the item store database.

To resolve this issue we applied a correction script to the captured requests to update references to missing product data to a newer available revision of the same data. The effect of updating these references was that the jobs associated with the affected requests were able to be executed again, though they would not necessarily run on the exact same data as they had in the production environment. The consequences of changing the input data for these jobs are further discussed in section 7.4.1.

For most affected requests a later revision of the product data was available, only for 90 of the captured requests no other version of the product data was available. For these requests no correction was applied and the requests were kept in the workload trace ‘as-is’. The effect of leaving requests as-is is that the jobs they trigger will always fail during the replay process: since this does not prevent other jobs from running and since this applies to only 90 requests this should not negatively affect the workload replay process.

8.2.2 Changed generation identifiers in item store objects

As described in section 2.4 the item store database contains references to the data files in the item store objects that are required to load a project’s item data. Because these files are periodically updated

and deleted asynchronously, multiple versions of the same file can exist. To disambiguate between versions of the same file each reference to a file contains the file's unique 'generation identifier', which is an arbitrary number that changes each time a new version of a file is uploaded.

We discovered that copying the item store objects also caused all generation identifiers to be reset to a new value, which invalidated all stored references in the item store. This in turn caused all rule processing jobs for projects that required item store objects (over 50% of all jobs) to be unable to run.

To resolve this issue we applied a correction script to the copied PostgreSQL database that updated the stored generation identifiers. The correct generation identifiers to use in our experiment were determined by making the script used to copy the files between GCS buckets also record the original generation identifier and a MD5 checksum of the copied file. By matching this information against the files in the copied GCS bucket a mapping from the original generation identifiers to new generation identifiers could be made, which we used to update the generation identifiers in the copied PostgreSQL database.

This change allowed all affected jobs to run in the experiment environment.

8.2.3 Outlier jobs unable to fit in hardware

We discovered that some of the jobs in the captured workload trace were so computationally intensive that they were unable to be executed on the rule processing servers in the experimental setup. When executed on one of these servers these jobs did not only fail to run, but also occasionally caused unrelated jobs that were running at the same time to be terminated as well.

This behaviour was not unexpected: in Channable's production environment these jobs were already assigned to a dedicated server with extra computational resources compared to a normal rule processing server.

To deal with this problem the affected jobs were removed from the captured workload trace. This removed 38 requests from the workload trace.

9 Evaluated scheduling approaches

In order to compare the effectiveness of different scheduling approaches we added support for multiple scheduling policies to the scheduler described in section 7.3.

Each scheduling policy consists of two parts: an *overload detection policy* and a *host selection policy*.

When a scheduling request is received by the scheduler, the scheduler performs the following steps:

1. Query the service discovery system to determine which servers exist and are able to accept rule processing jobs.
2. Apply the configured overload detection policy to exclude servers that the policy considers to be overloaded.
3. If there are no non-overloaded servers currently available, return an error to the client indicating it should retry later.
4. Otherwise, consult the host selection policy to determine which of the available rule processing servers the request should be routed to.

In this section we describe the overload detection and host selection policies that we implemented in our scheduler.

9.1 Overload detection policies

The scheduler uses an overload detection policy to determine whether a rule processing server should receive new rule processing jobs. This approach allows the scheduler in our experiment environment to stop scheduling rule processing jobs on servers before the server is totally overloaded, which is not currently possible in the production environment.

All overload detection policies require a rule processing server to be marked as 'healthy' in the service discovery system: if a server is unhealthy the server is either offline or so heavily overloaded that the monitoring system is unable to retrieve the server's metrics, so any rule processing jobs are unlikely to be accepted.

Our scheduler implements three overload detection policies on top of this requirement, which we describe below.

9.1.1 Baseline overload detection

The baseline overload detection policy mimics the overload detection method in use in the production environment, which is no overload detection at all.

As described in section 4.1.2 the production environment does not perform any additional checks beyond checking that the rule processing servers are listed as healthy in the service discovery system. To mimic this behaviour in our scheduler the baseline overload detection policy is a no-op: all healthy servers are considered eligible to receive new rule processing jobs.

9.1.2 Circuit breaker

The circuit breaker overload detection policy uses a static circuit breaker as described by Sedghpour, Klein, and Tordsson [9], using the 'stalled promises' metric described in section 3.4 as the trigger.

Under the circuit breaker overload policy a server is immediately considered ineligible to receive jobs if the server is unhealthy (as in the baseline policy) or if the amount of stalled promises exceeds a

configurable threshold. A high number of stalled promises indicates that the rule processing server is CPU-starved and has a backlog of work it hasn't executed yet, so stopping scheduling new jobs on the server should allow it to process its backlog, or at least prevent the backlog from growing further.

To implement this scheduling policy, the scheduler queries the monitoring system every five seconds to retrieve the most recent reported amount of stalled promises for each rule processing service. This amount is cached in the scheduler's information for each server.

When a scheduling request arrives, the scheduler compares the most recently known amount of stalled promises against the configured threshold and excludes all servers where this amount exceeds the configured threshold.

The effectiveness of this policy depends on what threshold of stalled promises is configured in the scheduler: if a too-low threshold is chosen we expect that servers would be under-utilized, while a too-high threshold would be ineffective in protecting servers from being overloaded. To determine an appropriate threshold for the circuit breaker we performed a series of tests where we replayed only the first 10 minutes of the captured workload trace and determined whether any rule processing servers were overloaded. We initially started these tests with a threshold of 400 stalled promises. This starting point was chosen because this level is currently used by Channable's operations team to trigger an alert for manual investigation.

We repeated the 10-minute tests with lower thresholds until no rule processing servers became overloaded during the tests, which occurred at a threshold of 100 stalled promises. Because of this we chose 100 stalled promises as our threshold for the circuit breaker overload detection policy.

9.1.3 DAGOR

The DAGOR overload detection policy uses the DAGOR load shedding method as described by Zhou et al. [14] to gradually shed load from overloaded rule processing servers.

As described in section 5.4 Zhou et al. used the average waiting time of requests in a server's queue to determine whether or not a server is overloaded. While implementing this policy we discovered that using the average waiting time is unfortunately not currently possible for the rule processing service: the executor does have information on the average queuing time of its rule processing jobs, but it only exposes the queuing time of jobs that have already completed, and not the queuing time of jobs that have yet to be executed. Exposing the actual waiting time of the rule processing jobs in a server's queue should technically be possible, but is outside the scope of this project.

To work around this missing information we decided to use the number of stalled promises as the DAGOR algorithm's input to determine overload, instead of the queuing time. This makes the DAGOR overload detection policy similar to the circuit breaker overload detection policy, but rather than immediately excluding a server from receiving new requests it should gradually reduce the amount of requests sent to the server while it is overloaded.

We configured the scheduler to use the same threshold as used by the circuit breaker policy (100 stalled promises) for the DAGOR policy's overload threshold.

9.2 Host selection

Once the scheduler has determined the set of rule processing servers that are eligible to receive new rule processing jobs it applies a host selection policy to determine which of the selected hosts should receive the rule processing job.

The earlier project by Ulita [11] focused on assigning rule processing jobs to hosts using a 'caching plan', a pre-computed static assignment of projects to rule processing servers. We have made the decision not to continue with this approach.

Ulita's caching plan approach assigns projects to specific rule processing servers, falling back to selecting a random server if a selected server is unavailable. When Ulita performed his project Channable did not yet use preemptible servers (see section 3.6) for the rule processing servers and all rule processing servers could be assumed to only be unavailable for short periods of time, e.g. for software upgrades. Because the production environment's rule processing servers are now (partially) deployed on preemptible servers, this is no longer the case: servers are regularly offline for periods of five to ten minutes, which can occur at any time.

We expect that the use of preemptible servers makes the use of a caching plan less effective, as the scheduler will be forced to fall back to selecting a random server more often than when non-preemptible servers are used.

Our scheduler instead implements host selection using a new approach, in addition to two approaches already in use. These host selection policies are described in this section.

9.2.1 Baseline host selection

The baseline host selection policy mimics the host selection method used in production, as described in section 4.1. Under the baseline host selection policy, the scheduler selects a host by:

1. Ordering the set of eligible hosts using a fixed order (lexographically by server name),
2. Initializing a pseudo-random number generator using the job's project identifier as the seed,
3. Randomizing the order of the list of eligible hosts using this pseudo-random number generator,
4. Returning the first host in the resulting list.

Like the production environment's host selection, this host selection policy is deterministic (even though it uses a pseudo-random number generator): given the same set of available servers and project ID it will always select the same server.

9.2.2 Random host selection

The random host selection policy does not use any of the data present in the scheduler, and instead picks a random eligible host to execute the job on. Contrary to the baseline host selection policy, which also picks a pseudo-random host, the random host selection policy is not deterministic: it may pick a different server for each request.

This host selection policy is only used to select hosts for rule processing jobs that don't depend on project data: in these cases it does not matter on which host the job is executed.

9.2.3 Dynamic host selection

The dynamic host selection policy uses the cache information available in the scheduler to select which host a rule processing job should be executed on.

Under the dynamic host selection policy, the scheduler selects a host by ordering the set of available hosts using two metrics:

1. The 'cache status' of the project, i.e. whether the scheduler expects the project's data to be in cache on the server being considered. This can be one of three values, in descending order of preference:
    - 'Known cached', indicating the server reported having the project in cache,

- 'Assumed cached', indicating the server did not report the project's data as being cached, but the scheduler recently assigned a job for the same project to the server. In this case the scheduler assumes that the data is either already in cache but not yet known to the scheduler, or that the rule processing server will be able to apply de-duplication as described in section 3.5 to only fetch the relevant project data a single time.
- 'Not cached', indicating the scheduler does not expect the server to have the project in cache.

2. The number of rule processing jobs that the scheduler estimates to be in progress on the host. This is defined as the last known amount of running rule processing jobs, plus the amount of rule processing jobs the server assigned to the host since the amount of running jobs was last retrieved.

This ordering makes the policy effectively prefer hosts where the project's data is in cache or can be assumed to be in cache, preferring servers with less rule processing jobs in progress if the project has the same cache status on multiple servers.

9.3 Experiments

To evaluate the scheduling policies described in this section we performed an experiment for each combination of overload detection policy and host selection policy, excluding the random host selection policy. This resulted in a set of six experiments.

Table 3 shows the names we will use to refer to each of the experiments in later figures.

Overload detection policy	Host selection policy	Experiment name
Baseline	Baseline	Baseline, baseline
Circuit breaker	Baseline	Breaker, baseline
DAGOR	Baseline	DAGOR, baseline
Baseline	Dynamic	Baseline, dynamic
Circuit breaker	Dynamic	Breaker, dynamic
DAGOR	Dynamic	DAGOR, dynamic

Table 3: Overview of the experiments we performed and the names with which we refer to them.

10 Results

In this section we present the results of the experiments we describe in section 9.3. We compare the performance of the rule processing jobs in each experiment to the baseline experiment and the production environment's performance.

10.1 Server preemption oversight

As described in section 7.4.4 we set up our experiment environment using preemptible servers for all rule processing servers. We had expected this change to have a minimal impact to the behaviour of our experiment environment, but due to an oversight this change did cause an error in the experiment environment.

Before starting an experiment we restart all rule processing servers, to ensure that the rule processing servers are in a known state. We failed to anticipate that this would also cause all rule processing servers to shut down at the same time, as the runtime of preemptible servers is limited to 24 hours (see section 3.6).

This is highly undesirable: because the experiments last for slightly more than 24 hours this effectively caused a period in each experiment where no rule processing servers were available. The production environment does not suffer from this problem, because there the preemptible servers are started at different times of the day and there are also non-preemptible servers that are not automatically shut down.

The ideal way to deal with this issue would be to re-run the experiments while preventing the servers from restarting all at once, which could be done by either using non-preemptible servers (at an increased cost), or by starting each of the servers at a different time of day (e.g. with about 10 minutes of spacing between them). This should prevent all servers being restarted at the same time.

We were unable to rerun all experiments due to time constraints, as we unfortunately discovered this issue only after all experiments had been run. Instead of rerunning all experiments we decided to truncate the results of each experiment at the point where the shutdown occurred, to remove the effect of the shutdown period from our data. To ensure that the results of each experiment remained comparable we use the same cutoff point for all experiments' data, the earliest shutdown moment across all experiments. This shutdown occurred 22 hours and 48 minutes after the start of the experiment. To compensate for our error we exclude any rule processing jobs which completed after 22 hours and 48 minutes from our data.

10.2 Job duration

Figure 4 shows the mean duration of the rule processing jobs across all servers for the production environment and each of our six experiments. Figures 5 and 6 show the distributions of these durations in each experiment by comparing the median duration and the 90^{th}, $95^{th}$and 99^{th}percentiles of the durations. The data that these graphs visualize is shown in the appendix in table 4.

The high-priority jobs in our experiments all seem to perform comparably to the production environment: the mean duration and percentiles do not change significantly from the duration in the production environment.

The durations of the low-priority jobs do change significantly. In the baseline experiment the duration of the low-priority jobs seems to degrade significantly: the mean duration of the low-priority jobs is double the mean duration in the production environment. This seems to be mainly caused by an increase in duration of the worst-performing jobs: the $90^{th}$and 95^{th} percentile of the jobs' durations is more than triple the value in the production environment, while the median duration does not change significantly.

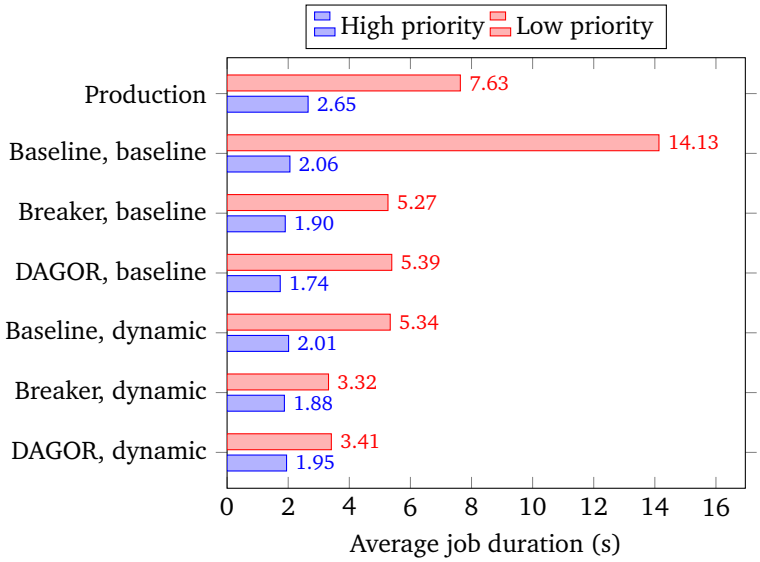

Figure 4: Mean durations for rule processing jobs per priority per experiment.

All other experiments perform significantly better than both the production environment and baseline experiment: the experiments where only the overload detection policy or the host selection policy are changed have a mean duration 2.6 times lower than the baseline experiment and 1.4 times lower than the production environment. The experiments where both the overload detection policy and host selection policy are changed perform even better, with a mean duration 2.2 times faster than the production environment and 4.1 times faster than the baseline experiment. The outlier percentiles of the jobs' durations also improve, though the median duration of the low-priority jobs again does not change significantly.

Over all experiments the experiment using the circuit breaker overload detection policy and dynamic host selection policy seems to perform the best, with the experiment using DAGOR overload detection and dynamic host selection as a close second-best.

10.3 Waiting time

Figure 7 shows the mean waiting time of the rule processing jobs across all servers for the production environment and each of our six experiments. Figures 8 and 9 show the distributions of the waiting time in each experiment by comparing the median waiting time and the 90^{th}, $95^{th}$and 99^{th}percentiles of the waiting time. The data that these graphs visualize is shown in the appendix in table 5.

The waiting time shows similar behaviour to the duration. For high priority jobs the mean waiting time is roughly equal across all experiments, and the median and even 90^{th} percentile of the waiting time are close to zero.

We do see significant changes in the waiting time for low-priority jobs. Like with the duration the waiting time seems to increase significantly in the baseline experiment as compared to the production environment: the mean waiting time is more than doubled, and the outlier percentiles also all perform significantly worse, though the median waiting time is unaffected.

We also see similar improvements in waiting time for the low-priority jobs in the non-baseline experiments as we saw for the duration: in the experiments with only a non-baseline host selection or overload detection policy the mean waiting time is 3.5 times lower than the baseline experiment and

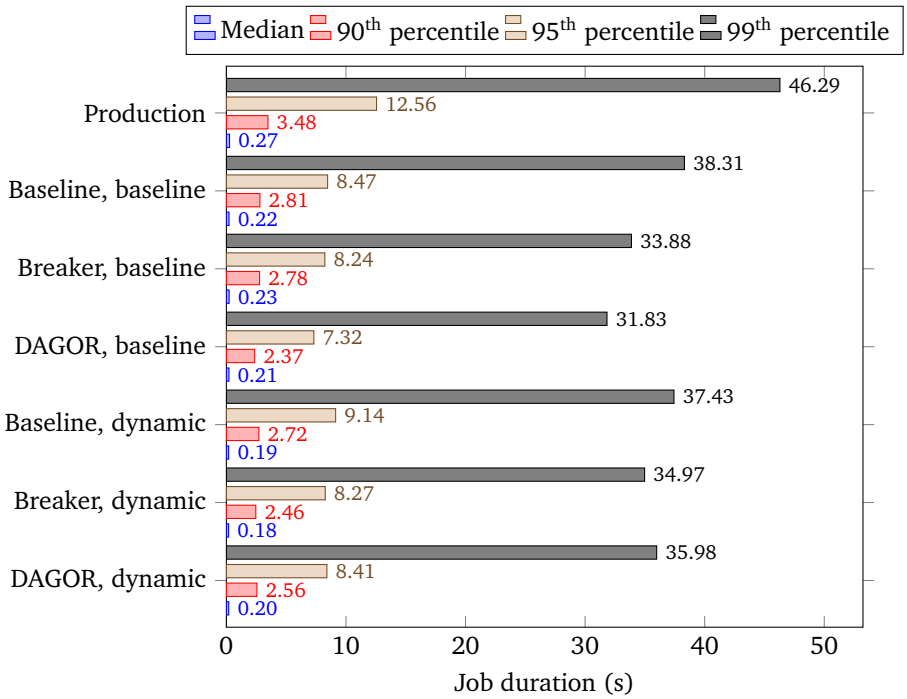

Figure 5: Percentile overview of durations for high-priority jobs per experiment.

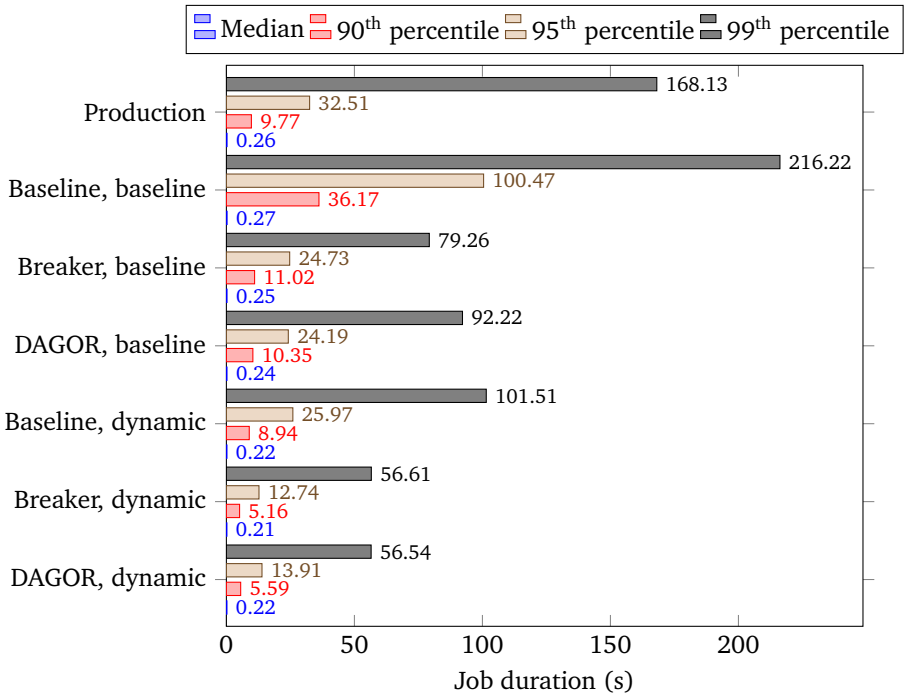

Figure 6: Percentile overview of durations for low-priority jobs per experiment.

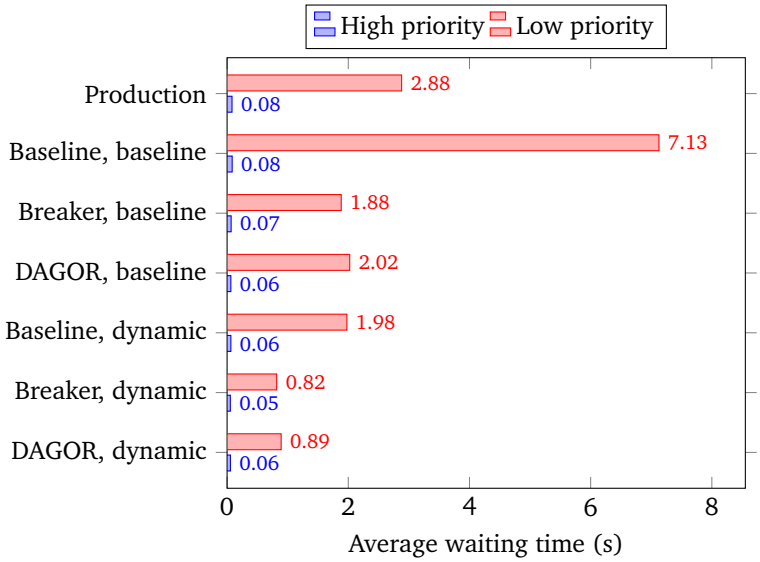

Figure 7: Mean rule processing job waiting times per priority in the production environment and in our experiments.

1.4 times lower than the production environment. The experiments with both a non-baseline overload detection and host selection policy again perform the best, with a mean waiting time 3.2 times lower than the production environment and 8 times lower than the baseline experiment.

The median waiting time of the low-priority jobs is unaffected and in fact almost zero in both the production environment and all experiments, similar to the waiting time for the high-priority jobs. We also note that the worst-case percentiles of the waiting time for low-priority jobs seems to only decrease if the dynamic host selection policy is used: the 90^{th} and 95^{th} percentiles of the wait time for the circuit breaker and DAGOR experiments with baseline host selection are similar to the production environment's performance.

The best-performing experiments with regards to waiting time are again the experiments that use both a non-baseline overload detection policy and the dynamic host selection policy.

Next to the absolute waiting times we also inspected the variance in waiting time for low-priority jobs between the various servers.

Figure 10 shows the variance in wait time for low priority jobs for the production environment and each of the experiments. Table 6 shows the data that this graph represents.

We note that the variance in wait time varies wildly between servers in both the production environment and the baseline experiment: we see that some servers have an abnormally high variance in the waiting times, while other servers have a near-zero variance in waiting time. In our other experiments we see a more evenly distributed variance, where the average variance for the servers is both lower and more evenly distributed. The experiment using the circuit breaker overload detection policy in combination with the dynamic host selection policy seems to perform the best: the variance across all servers is the lowest and there are no outliers with a significantly higher or lower variance in this experiment.

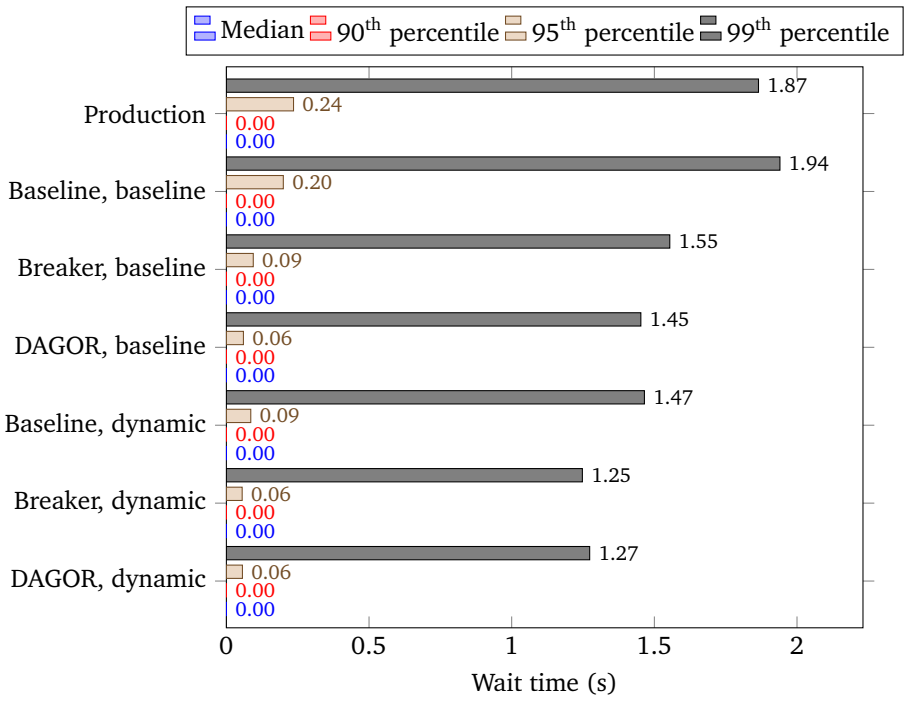

Figure 8: Percentile overview of waiting time for high-priority jobs in the production environment and our experiments.

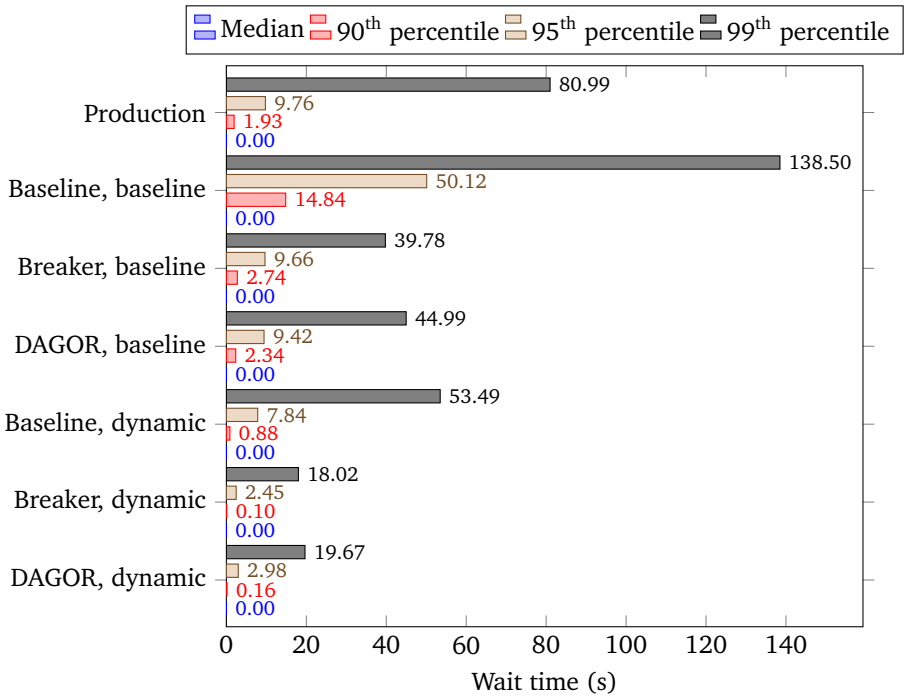

Figure 9: Percentile overview of waiting time for low-priority jobs in the production environment and our experiments.

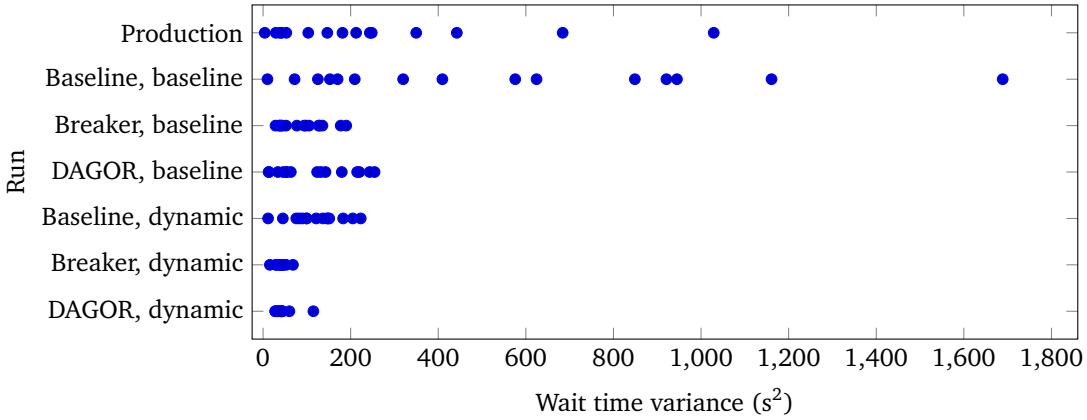

Figure 10: Variance in wait time for low-priority jobs per rule processing server in the production environment and experiments. Each dot represents the variance in wait time for one server.

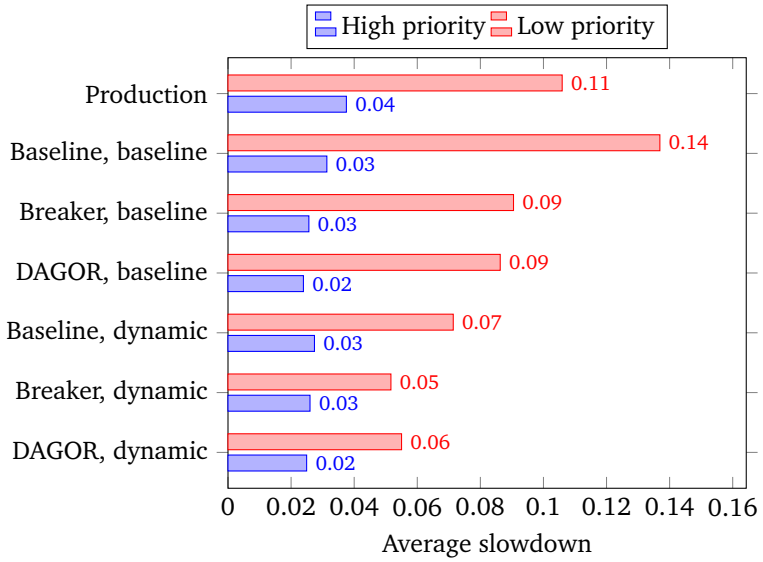

Figure 11: Mean slowdown for rule processing jobs per priority per experiment.

10.4 Slowdown

Figure 11 shows the mean slowdown of the rule processing jobs across all servers for the production environment and each of our six experiments. Figures 12 and 13 show the distributions of the slowdown by comparing the median slowdown and the 90th, 95thand 99thpercentiles of the slowdown. The data that these graphs visualize is shown in the appendix in table 7.

As explained in section 5.1 the slowdown of a rule processing job is defined as defined as $\frac{W}{W+E}$, where W is the job's waiting time and E is the job's execution time. Because the slowdown depends on the waiting time it is no surprise that we see a similar pattern in the mean and worst-case percentiles of the slowdown as we saw for the waiting time statistics.

The slowdown for high-priority jobs seems to be largely unaffected by our experiments. The mean and 99th percentile of the slowdown in our experiments are not significantly different from the value in the production environment, and like the waiting time the median and 90th percentile of the slowdown are close to zero. The 95th percentile of the slowdown for high-priority jobs does decrease: it is about 4.4 times lower for the best-performing experiment (circuit breaker with dynamic host selection) compared to the production environment's performance.

For the low-priority jobs there does appear to be a significant decrease in slowdown, similar to the decrease in waiting time. We note that the mean slowdown in the best-performing experiment is half the mean slowdown in the production environment. The median slowdown is unaffected since it was close to zero already, as the median waiting time is also close to zero. The 90th and 95th percentiles of the slowdown also improve, but not the 99th percentile, which remains high.

10.5 Other observations

10.5.1 Bad performance in baseline experiment

In the previous sections we noted that the performance of low-priority jobs seems to be significantly worse in our baseline experiment when compared to the production environment, as the mean waiting

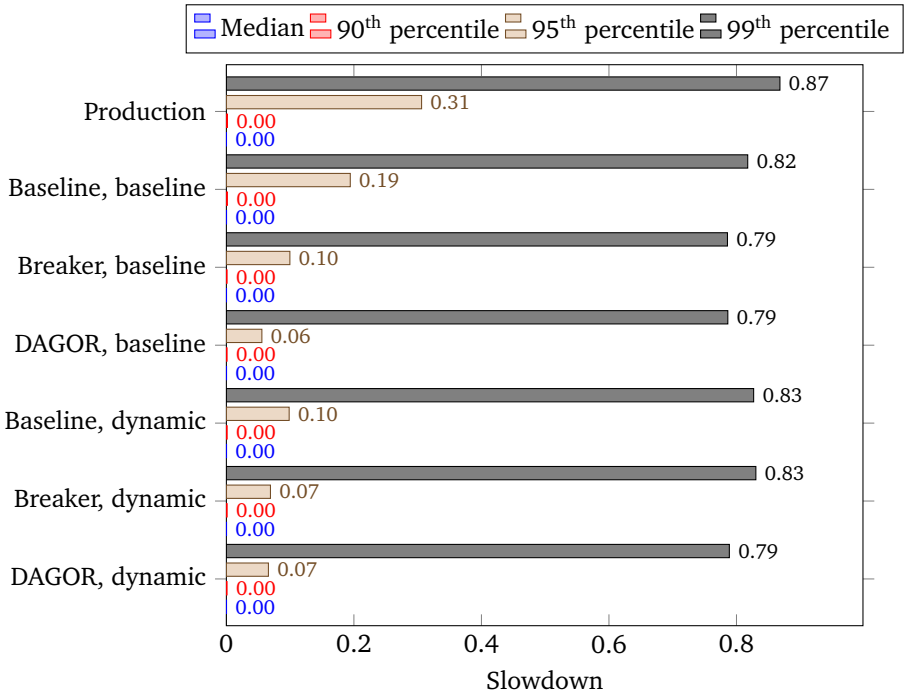

Figure 12: Slowdown percentiles for high-priority jobs, per run.

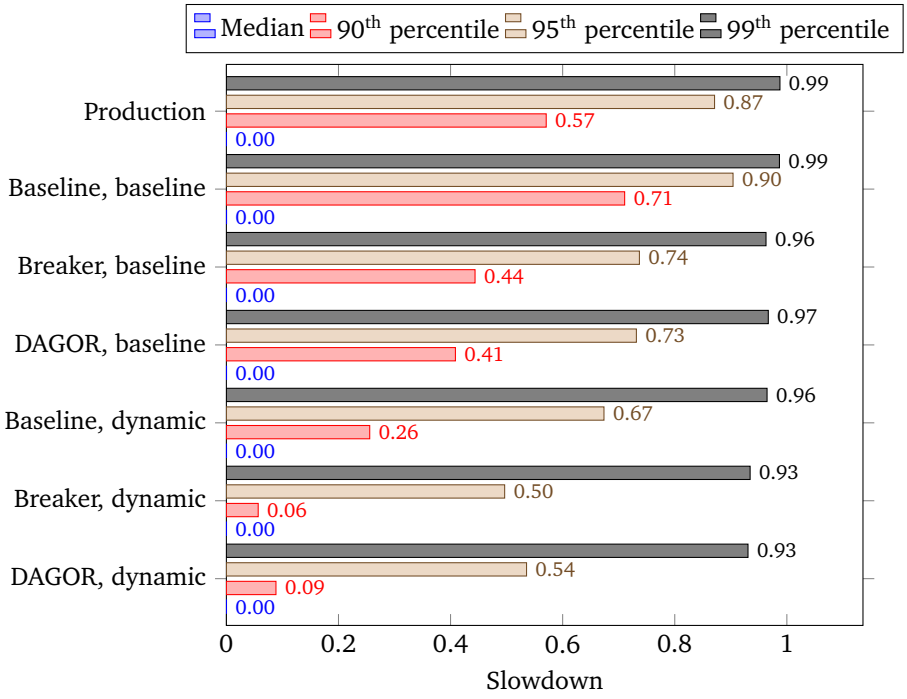

Figure 13: Slowdown percentiles for low-priority jobs, per run.

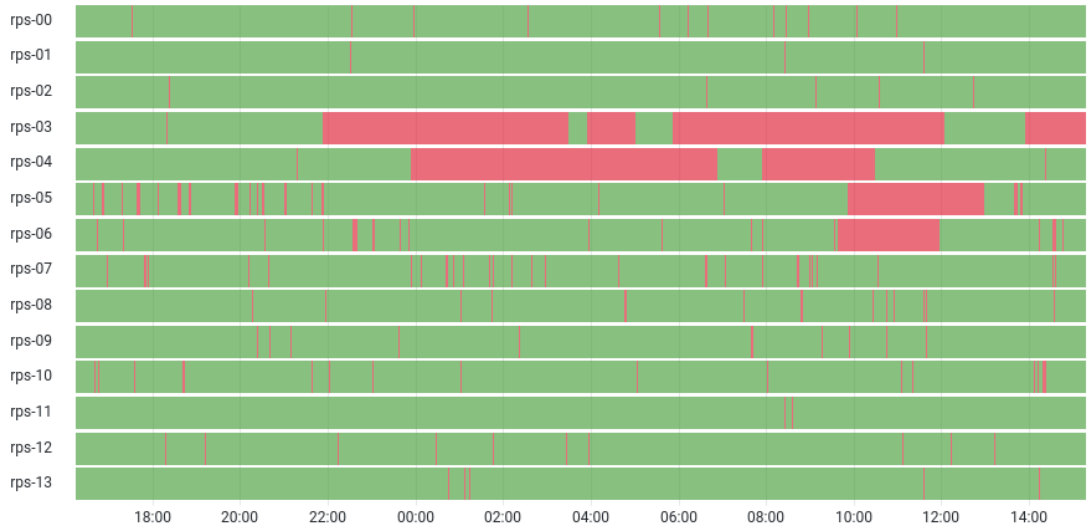

Figure 14: Availability history of the rule processing servers during the baseline experiment. A green bar indicates the rule processing server was healthy during this time, while a red bar indicates the rule processing server was overloaded.

time and duration of the low-priority jobs are both more than doubled compared to the production environment, and the outlier percentiles for these metrics also increase.

We also note that during the baseline experiment some of the rule processing servers behaved significantly worse than their counterparts in the production environment. Figure 14 shows the availability history of the servers during the baseline experiment. The figure shows that some of the servers were overloaded for significant amounts of time, which meant these rule processing servers were unable to accept new rule processing jobs. The worst performing server 'rps-03' was overloaded for more than 12 hours, and the second-most overloaded server 'rps-04' was overloaded for more than 9 hours.

We only observed this behaviour during the baseline experiment: during the other experiments rule processing servers also became overloaded, but only for short periods of time, recovering automatically. This behaviour, which is also typical in the production environment, can also be seen for the other servers in figure 14.

This is unexpected, as we had meant to construct the experiment environment to mimic the production environment, so we also expected the baseline experiment to perform similarly to the production environment.

We are not sure what causes this difference in performance, but suspect it might be caused by the difference in timeout behaviour described in section 7.4.3. We believe that the lack of connection timeout might be causing the servers to become overloaded sooner: when a rule processing server is under load (but not yet completely overloaded) the time needed to accept new connections to the rule processing server's API increases. In the production environment this increase in connection time causes submitters to attempt to connect to different rule processing servers, as the submitters will trigger the connection timeout. The overloaded server's workload decreases as submitters fall back to different rule processing servers, since the server does not receive any new rule processing jobs anymore. This allows the server to recover from the overload condition.

As described in section 7.4.3 the submitter in the experiment environment does not have a

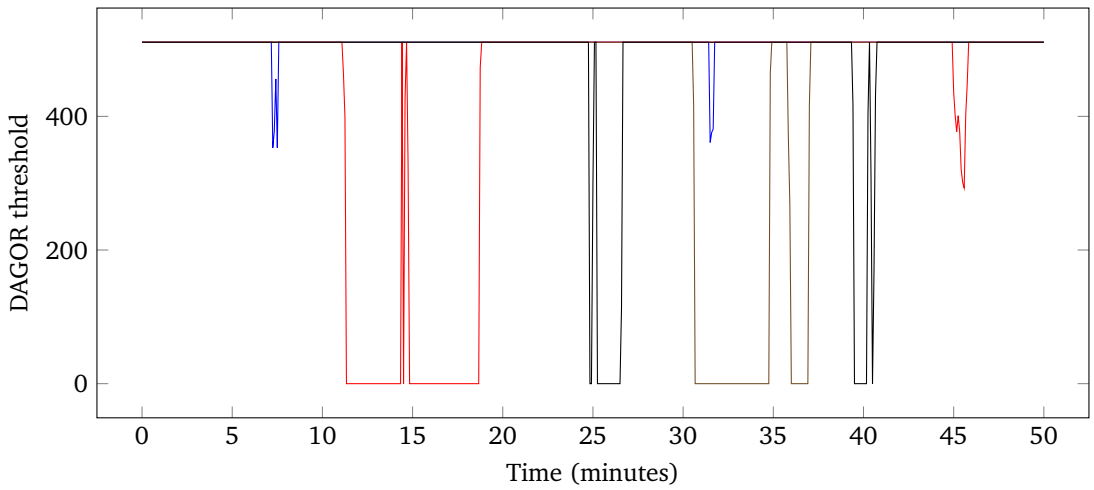

Figure 15: DAGOR threshold over time for four servers over 50 minutes of the experiment using DAGOR and baseline host selection. A high threshold means that the server is able to accept all jobs, while a low threshold indicates the server is overloaded.

connection timeout, which causes it to wait for the connection to be accepted for a longer period than a submitter in the production environment would wait. This means that the server's workload does not decrease, and that the server remains overloaded.

10.5.2 'Flip-flopping' in DAGOR experiments

We observed that in most experiments the circuit breaker overload detection policy outperforms the DAGOR overload detection policy by achieving lower request durations and wait times than the DAGOR overload detection policy.

On closer inspection we noticed that the DAGOR algorithm does not entirely seem to be behaving as intended: it appears that servers are sometimes rapidly switching ('flip-flopping') between the completely available state and a state of full overload when the servers are just on the threshold of being overloaded for an extended period of time. Figure 15 shows an example of this behaviour: in this example the server indicated by the blue line behaves 'correctly' by only briefly reducing the threshold, while the servers indicated by the red, black and brown lines start rejecting all load when they become overloaded, and then rapidly switch between the available state and the overloaded state after a short period.

This behaviour happens relatively often, which indicates that the DAGOR algorithm is not reaching its full potential, as it was specifically designed to perform gradual load shedding instead of making a binary overloaded/not overloaded decision.

We believe that this behaviour shows that DAGOR overload control might not be a good fit for situations where the average queuing time of requests in a server's queue is not available. As described in section 9.1.3 this information was not available in the rule processing service, and we instead opted to use the amount of stalled promises, as the DAGOR algorithm can be adjusted to use a different overload indicator. The DAGOR overload control policy might perform better if the average request queuing time were made available.

11 Conclusion

We originally set out to answer our research question:

> *How to dynamically schedule rule processing jobs to evenly divide the workload among a number of rule processing servers?*

To answer our research question we have implemented a centralized scheduler for the rule processing servers and compared three methods of overload detection for the rule processing servers, as well as two methods of host selection. We were also able to improve the process for capturing a workload trace (as previously used by Ulita), allowing us to obtain a more consistent workload trace, which allowed us to successfully replay a higher percentage of rule processing jobs in our experiment environment compared to Ulita's experiments.

Using our experiments we were able to determine that our scheduling approach using a static circuit breaker for overload detection and our dynamic host selection policy performs the best out of all evaluated scheduling approaches, significantly outperforming both our baseline experiment and the production environment's performance on the same jobs. While our baseline experiment did perform worse than the production environment we do believe that implementing our scheduler in the production environment will also lead to a comparable performance improvement over the current scheduling approach.

We believe this adequately answers our research question.

12 Future work

In this section we identify possibilities for improvement and future work.

12.1 Implementation in production

Given the results of our experiments we believe that our scheduling policy would lead to a performance increase in Channable's production environment, by resolving the performance issues associated with the current method of scheduling work on the rule processing servers. As such we recommend that Channable implements our scheduler in its production environment.

Because our experiment environment mimics the production environment we believe that it should be relatively straightforward to adapt our scheduler to be able to run in the production environment. We also do not foresee any major issues in adapting the submitter systems to query the scheduler for a rule processing server instead of making the scheduling decision themselves: since the scheduler offers a HTTP API and many other components of the feed processing system also already use HTTP APIs it should be easy to add support for our scheduler.

12.2 Determine impact of using a connection timeout

As noted in section 7.4.3 we discovered a difference between our experiment environment and the production environment, in that the experiment environment did not make use of connection timeouts. We note in section 10.5.1 that this difference might have caused our baseline experiment to behave worse than the production environment.

We believe it might be interesting to investigate the effect of reinstating the connection timeout, as it might be usable as an early warning for rule processing servers becoming overloaded. In order to investigate this effect a new experiment would need to be run, though the existing workload trace data can be reused.

If the experiment were to be repeated we also suggest addressing the problem with our use of preemptible infrastructure described in section 10.1. As described we believe it should be possible to address the issue by starting the rule processing servers one by one instead of all at once, so that the rule processing servers don't all end up being preempted at the same time.

12.3 Alternative scheduling approaches

Lastly, we suggest that more scheduling approaches could be evaluated. Because our scheduler is configurable it is relatively easy to add more overload detection policies or host selection policies to it, so this could provide an opportunity for future research.

As noted in section 9.1.3 we had to adjust the DAGOR overload detection policy, as the metric that Zhou et al. [14] use was not available for the rule processing servers. If the rule processing servers were adapted to expose the average waiting time of requests in the rule processing servers' queues one interesting research opportunity would be to re-implement the DAGOR algorithm using the 'proper' metric, which might allow for a further performance improvement.

The scheduling policies we evaluated also did not take all factors into account that we identified in section 4.3. It might also be interesting to implement alternative host selection policies or to extend the current host selection policy to take more factors of the rule processing servers' states into account, such as whether a server will be preempted soon, or the historical performance of a server.

References

[1] Ganesh Ananthanarayanan et al. “PACMan: Coordinated Memory Caching for Parallel Jobs”. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). 2012, pp. 267–280. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan.

[2] Eric Boutin et al. “Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing”. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014, pp. 285–300. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

[3] Marc Brooker. *Exponential Backoff And Jitter*. AWS Architecture Blog. Section: Architecture. Mar. 4, 2015. URL: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/ (visited on 02/10/2023).

[4] Inho Cho et al. “Overload Control for µs-scale RPCs with Breakwater”. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020, pp. 299–314. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/cho.

[5] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Communications of the ACM* 56.2 (Feb. 1, 2013), pp. 74–80. ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. URL: https://doi.org/10.1145/2408776.2408794.

[6] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. “Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’19. New York, NY, USA: Association for Computing Machinery, Nov. 20, 2019, pp. 233–245. ISBN: 978-1-4503-6973-2. DOI: 10.1145/3357223.3362724. URL: https://doi.org/10.1145/3357223.3362724.

[7] Makoto Matsumoto and Takuji Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (Jan. 1, 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: https://doi.org/10.1145/272991.272995.

[8] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. “When to Hedge in Interactive Services”. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). 2021, pp. 373–387. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/primorac.

[9] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. “Service mesh circuit breaker: From panic button to performance management tool”. In: *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*. HAOC ’21. New York, NY, USA: Association for Computing Machinery, Apr. 26, 2021, pp. 4–10. ISBN: 978-1-4503-8336-3. DOI: 10.1145/3447851.3458740. URL: https://doi.org/10.1145/3447851.3458740.

[10] Guy L. Steele, Doug Lea, and Christine H. Flood. “Fast splittable pseudorandom number generators”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. New York, NY, USA: Association for Computing Machinery, Oct. 15, 2014, pp. 453–472. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660195. URL: https://doi.org/10.1145/2660193.2660195.

[11] Paweł Ulita. “Scheduling data processing tasks for product feed management”. Master’s thesis. University of Amsterdam, May 7, 2020.

[12] Matei Zaharia et al. “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling”. In: *Proceedings of the 5th European conference on Computer systems*. EuroSys ’10. New York, NY, USA: Association for Computing Machinery, Apr. 13, 2010, pp. 265–278. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. URL: https://doi.org/10.1145/1755913.1755940.

[13] Hao Zhou et al. “Overload Control for Scaling WeChat Microservices”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’18. New York, NY, USA: Association for Computing Machinery, Oct. 11, 2018, pp. 149–161. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267823. URL: https://doi.org/10.1145/3267809.3267823.

[14] Hao Zhou et al. “Overload Control for Scaling WeChat Microservices”. In: *arXiv:1806.04075 [cs]* (Dec. 23, 2018). version: 3. DOI: 10.1145/3267809.3267823. arXiv: 1806.04075. URL: http://arxiv.org/abs/1806.04075.

A Job duration statistics

	High priority						Low priority					
Run	**Mean**	**Median**	**p90**	**p95**	**p99**	**Max.**	**Mean**	**Median**	**p90**	**p95**	**p99**	**Max.**
Production	2.65	0.27	3.48	12.56	46.29	438.59	7.63	0.26	9.77	32.51	168.13	7,201.08
Baseline, baseline	2.06	0.22	2.81	8.47	38.31	361.64	14.13	0.27	36.17	100.47	216.22	8,721.52
Breaker, baseline	1.90	0.23	2.78	8.24	33.88	361.90	5.27	0.25	11.02	24.73	79.26	1,970.64
DAGOR, baseline	1.74	0.21	2.37	7.32	31.83	393.45	5.39	0.24	10.35	24.19	92.22	790.93
Baseline, dynamic	2.01	0.19	2.72	9.14	37.43	437.18	5.34	0.22	8.94	25.97	101.51	1,105.56
Breaker, dynamic	1.88	0.18	2.46	8.27	34.97	442.70	3.32	0.21	5.16	12.74	56.61	2,570.80
DAGOR, dynamic	1.95	0.20	2.56	8.41	35.98	433.26	3.41	0.22	5.59	13.91	56.54	930.36

Table 4: Duration statistics in seconds for the rule processing jobs executed in the production environment and each experiment. The columns labeled p90, p95, p99 represent the 90^{th}, 95^{th}, 99^{th} percentiles of the duration.

B Wait time statistics

	High priority						Low priority					
Run	**Mean**	**Median**	**p90**	**p95**	**p99**	**Max.**	**Mean**	**Median**	**p90**	**p95**	**p99**	**Max.**
Production	0.08	$5.00 \cdot 10^{-5}$	$1.20 \cdot 10^{-4}$	0.24	1.87	38.22	2.88	$4.00 \cdot 10^{-5}$	1.93	9.76	80.99	273.30
Baseline, baseline	0.08	$6.00 \cdot 10^{-5}$	$1.90 \cdot 10^{-4}$	0.20	1.94	36.06	7.13	$6.00 \cdot 10^{-5}$	14.84	50.12	138.50	242.19
Breaker, baseline	0.07	$6.00 \cdot 10^{-5}$	$1.10 \cdot 10^{-4}$	0.09	1.55	42.00	1.88	$5.00 \cdot 10^{-5}$	2.74	9.66	39.78	240.17
DAGOR, baseline	0.06	$6.00 \cdot 10^{-5}$	$1.10 \cdot 10^{-4}$	0.06	1.45	47.41	2.02	$6.00 \cdot 10^{-5}$	2.34	9.42	44.99	239.99
Baseline, dynamic	0.06	$6.00 \cdot 10^{-5}$	$1.10 \cdot 10^{-4}$	0.09	1.47	34.05	1.98	$5.00 \cdot 10^{-5}$	0.88	7.84	53.49	239.63
Breaker, dynamic	0.05	$6.00 \cdot 10^{-5}$	$1.00 \cdot 10^{-4}$	0.06	1.25	25.57	0.82	$4.00 \cdot 10^{-5}$	0.10	2.45	18.02	240.00
DAGOR, dynamic	0.06	$6.00 \cdot 10^{-5}$	$1.00 \cdot 10^{-4}$	0.06	1.27	27.72	0.89	$5.00 \cdot 10^{-5}$	0.16	2.98	19.67	239.56

Table 5: Wait time statistics in seconds for the rule processing jobs executed in the production environment and each experiment. The columns labeled p90, p95, p99 represent the 90^{th}, 95^{th}, 99^{th} percentiles of the wait time.

		Variance per server													
Run	**All**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Production	248.13	181.27	212.45	349.68	29.13	442.37	1,028.93	243.73	3.88	103.16	38.39	146.63	53.05	42.50	684.06
Baseline, baseline	624.39	944.92	10.04	1,688.71	319.81	1,160.91	209.35	71.95	920.90	152.67	409.40	125.12	170.18	575.71	848.86
Breaker, baseline	93.96	37.39	28.48	104.93	77.22	189.85	37.85	41.42	126.80	177.23	97.36	44.66	52.40	129.28	135.51
DAGOR, baseline	123.41	132.03	13.43	254.58	53.71	214.79	12.80	46.25	179.73	33.85	142.55	63.36	53.31	243.30	219.83
Baseline, dynamic	121.34	182.75	204.85	82.75	99.27	90.13	147.86	151.06	75.79	223.03	146.11	45.11	136.48	11.48	99.35
Breaker, dynamic	40.74	43.26	28.99	45.74	40.53	43.15	31.62	38.73	39.10	45.71	68.24	52.76	40.20	15.51	43.16
DAGOR, dynamic	43.78	43.79	32.51	39.31	60.08	42.14	27.56	43.19	42.42	33.84	40.82	28.18	32.26	39.23	114.86

Table 6: Variance of wait times over all servers and per server for the low-priority rule processing jobs executed in the production environment and each experiment. The 'All' column shows the variance across all servers, while the numbered columns show the variance for rule processing jobs executed on a single server.

C Slowdown statistics

	High priority						Low priority					
Run	**Mean**	**Median**	**p90**	**p95**	**p99**	**Max.**	**Mean**	**Median**	**p90**	**p95**	**p99**	**Max.**
Production	0.04	$1.80 \cdot 10^{-4}$	0.00	0.31	0.87	1.00	0.11	$2.10 \cdot 10^{-4}$	0.57	0.87	0.99	1.00
Baseline, baseline	0.03	$2.70 \cdot 10^{-4}$	0.00	0.19	0.82	0.99	0.14	$3.30 \cdot 10^{-4}$	0.71	0.90	0.99	1.00
Breaker, baseline	0.03	$2.30 \cdot 10^{-4}$	0.00	0.10	0.79	0.99	0.09	$2.50 \cdot 10^{-4}$	0.44	0.74	0.96	1.00
DAGOR, baseline	0.02	$2.60 \cdot 10^{-4}$	0.00	0.06	0.79	0.99	0.09	$2.90 \cdot 10^{-4}$	0.41	0.73	0.97	1.00
Baseline, dynamic	0.03	$2.60 \cdot 10^{-4}$	0.00	0.10	0.83	0.99	0.07	$2.60 \cdot 10^{-4}$	0.26	0.67	0.96	1.00
Breaker, dynamic	0.03	$2.60 \cdot 10^{-4}$	0.00	0.07	0.83	1.00	0.05	$2.30 \cdot 10^{-4}$	0.06	0.50	0.93	1.00
DAGOR, dynamic	0.02	$2.40 \cdot 10^{-4}$	0.00	0.07	0.79	1.00	0.06	$2.20 \cdot 10^{-4}$	0.09	0.54	0.93	1.00

Table 7: Slowdown statistics for the rule processing jobs executed in the production environment and each experiment. The columns labeled p90, p95, p99 represent the 90^{th}, $95^{\text{th}}$and 99^{th} percentiles of the slowdown.