Automated Benchmarking of Container Applications

Paulius Dilkas

22nd August 2019

1 Introduction

Before an application is deployed to a cloud, one needs to allocate resources for different parts of the system. But what is the right resource configuration for *my* application? How would it change if my application became more popular or I added some extra functionality?

We present a benchmarking system that can simulate the workflow of a range of distributed applications, measure various performance metrics, and determine whether the simulated performance matches our expectations. The system comes with several predefined configuration files that represent realistic workloads of common applications. Every aspect of each configuration file can be adjusted, and new configuration files can be created.

Our simulated applications consist of a collection of components together with a control server, all arranged as a directed acyclic graph (DAG). The *control server* is responsible for generating and sending *messages*. Messages travel via the edges of the graph. When a *component* receives a message, it simulates a predefined amount of work. Properties of this work can be customised separately for each component.

The components are implemented as `RichMapFunction`s from Apache Flink. The entire simulation system is packaged into several Docker containers and can be deployed on either an OpenShift cluster or a local MiniShift setup. Performance metrics are recorded using Prometheus, which can be managed separately or deployed together with the rest of the system.

2 Architecture and Implementation

In this section we describe implementation details of the system at its current state. During the first stage of the project, the system was deployed and tested on MiniShift—a single-node OpenShift implementation. While the intent is to progress to an OpenShift cloud, some of the current implementation details are specific to MiniShift.

In Section 2.1 we describe our initial Docker container-based configuration and how it was transformed into *manifestos*, i.e., OpenShift deployment configuration files. We also discuss modifications to the standard Prometheus deployment on MiniShift as well as configuration files that can be used to simulate various applications, add new performance metrics, etc. In Section 2.2 we dive into Java code to explain what happens during execution: how the Flink app interacts with the control server and simulates work and how performance metrics are tracked and recorded.

2.1 Deployment

Flink deployment consists of a JobManager that manages the work, one or more TaskManagers that execute tasks, and a command that tells the JobManager what to do. In order to deploy these services on OpenShift, we need to put each 'work unit' (something that can run on a separate node) in its own Docker container. We ended up using three simple Dockerfiles that were put on Docker Hub in order to make them easily accessible to both OpenShift and MiniShift:

- one for Flink TaskManagers and JobManagers that extends the `flink` Dockerfile with a new configuration file and support for Prometheus;
- one to initialise a sequence of experiments (this is run separately so that most other containers could be reused for many experiments without having to restart them[1]);
- and one to run the control server.

In order to have these Dockerfiles run on OpenShift, we need to make a few adjustments, since every container on OpenShift is run as an unspecified user belonging to the `root` group, and many applications are built to be run as a specific user instead. To fix this, one needs to change the permissions and group ownership of relevant directories on the file system (to `775` and `root` respectively).

A Docker Compose file can then be used to combine several Dockerfiles into a valid deployment configuration. In this file we define five services: `JobManager`, `TaskManager`, `Control`, `Start`, and `Prometheus`, establishing open ports as pictured in Figure 1. This file was then converted to OpenShift manifestos using Kompose[2]. The generated manifestos, relevant network connections, and other dependencies are displayed in Figure 1. A notable difference between the two configurations is that while a Docker Compose file defines only services, OpenShift has both services and pods. *Service* manifestos define the network interface (i.e., what ports are open), while manifestos for *pods* contain the details of what Docker containers should be run, restart policy, additional data that should be mounted to the pod, etc. The entire system can then be updated and deployed by generating a new JAR file using Maven, building and uploading the Dockerfiles, and recreating all components of the OpenShift configuration, as described by the manifestos.

Configuration Files component in Figure 1 represents a `ConfigMap` OpenShift entity created using the `oc` command that contains two configuration files: `global.yaml` and `components.yaml` (see Figures 2 and 3 for examples). The former contains some basic networking information such as hostnames and port numbers as well as:

`prometheusUsesHttps:` a Boolean variable indicating whether Prometheus is configured to provide data using HTTP or (broken) HTTPS. The MiniShift add-on uses HTTPS, while a run-of-the-mill Prometheus Docker container uses HTTP.

`numExperiments` defines how many (identical) experiments to run.

`delayBetweenExperiments` defines how long to wait between experiments (in min). It is recommended to set this to about 1 min so that Prometheus data for one experiment would not contain data from the previous experiment.

`metrics` is a list of metrics, each described using three properties:

`name:` the 'pretty' name of the metric, used for plotting.

`filename:` the 'basic' name of the metric, used as part of the filename when saving performance data to a file.

`query:` the name of the metric as understood by Prometheus.

`workload` defines how many messages to send, and at what intervals. More specifically, it contains:

`messagesPerSecond:` how often to send messages.

`experimentDuration:` how long the experiment should last.

`requestsPerMessage:` how many messages/requests to send at a time.

[1] Restarting something on an OpenShift cluster often requires one to wait 3–5 min between some commands, otherwise networking management bugs out and services become unreachable.

[2] `http://kompose.io/`

The `components.yaml` configuration file, on the other hand, describes a sequence of processing stages (maps), each with its own CPU usage time, memory usage, and output data size (i.e., the amount of data passed to the next stage).

Lastly, it is worth mentioning that the Prometheus add-on for MiniShift[3] had to be modified in order to disable OAuth-based authentication by replacing

`-skip-auth-regex=^/metrics` with `-skip-auth-regex=^/`.

Furthermore, Prometheus configuration file was updated to set both scrape and evaluation intervals to 1 s and the list of targets to JobManager and TaskManager, both on port 9250.

2.2 What Happens During Execution

We illustrate some aspects of the execution and how different components communicate with each other in Figure 4. After the Flink app (called Benchmarker) is initialised, it immediately establishes the control server as a `socketTextStream`, i.e., the initial source of data. It then constructs the DAG of components as described in `components.yaml`.

The control server periodically sends messages to Benchmarker (as defined in `global.yaml`). Each component (mapper) does three things upon receiving each message:

1. First, it allocates an array of bytes so that the total memory usage would be as close to `memoryUsage` as possible. The array size is calculated using a linear regression model established using experimental data (see Section 3).

2. Then, it creates a `String` object taking up `outputSize` KiB of memory. This string will be passed to the next component in the chain.

3. Finally, it spends the remaining time (until total execution time is exactly `cpuTime`) testing the Collatz conjecture [5] one initial integer at a time.

After all messages from the control server pass through every component, Benchmarker connects to the control server, sending it the total running time (as measured by `JobExecutionResult.getNetRuntime()`). This number is then rounded up to an integer number of minutes and used to retrieve performance data for the time interval when the application was running.

Finally, for each metric defined in the global configuration file, the control server establishes an HTTPS connection to Prometheus, collects JSON data recording the values of that metric in the last few minutes (as calculated previously), and writes the data to a file (separate for each metric) on the persistent volume. The files can then be transported to a local directory by using MiniShift SSH to copy them over to MiniShift host folder, which places them into a local directory on the host machine. A Python script was written to automate deploying the system, waiting for the control server to terminate, and moving the files as described.

3 Local Performance Tuning

The component class, responsible for using predefined amounts of resources, was tested and adjusted locally, ensuring that it uses 100% of a single CPU and `memoryUsage` MiB of memory. Total heap usage was measured for array sizes $2^0, 2^1, 2^2, \ldots, 2^9$ and output strings of $2^0, 2^1, 2^2, \ldots, \min\{2^8, \text{array size}\}$ characters (the output string is constructed using the array, so the array size must always be at least as big as the output string). Maximum heap usage was measured using GNU Time[4] and its Maximum Resident Set Size metric. Each experiment was repeated three times, and median values were taken.

[3] `https://github.com/minishift/minishift-addons/tree/master/add-ons/prometheus`
[4] `https://www.gnu.org/software/time/`

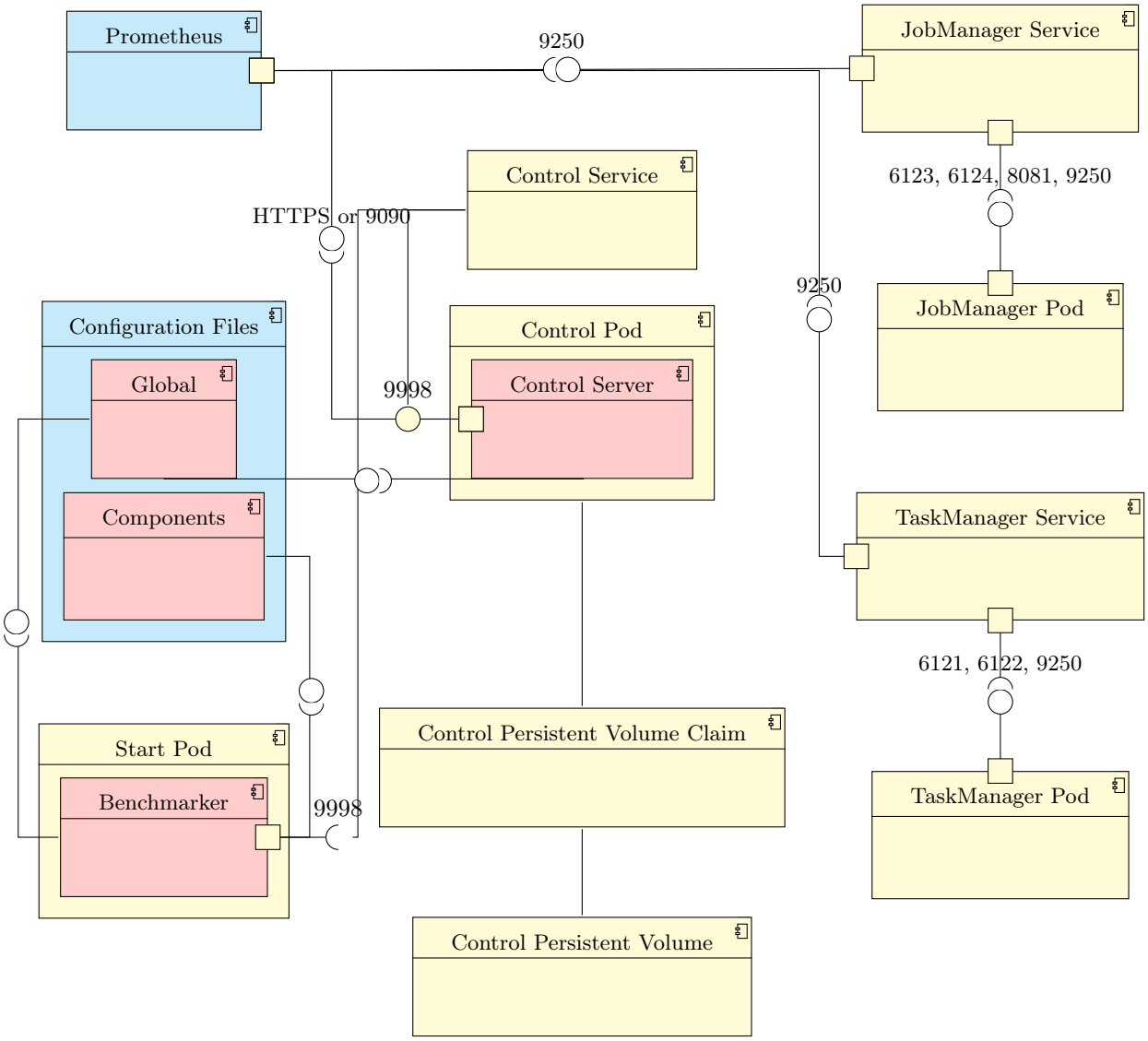


Figure 1: UML component diagram of the system. Yellow components are OpenShift manifestos, red components represent files (either Java classes or YAML configuration files), and cyan components are for everything else (components set up without explicit manifestos, etc.). Network connections are shown with ports and have port numbers (or application-layer protocol names) displayed.

```yaml
controlHostname: control
controlPort: 9998

prometheusHostname: prometheus
prometheusPort: 9090
prometheusUsesHttps: true

numExperiments: 1
delayBetweenExperiments: 1 # in min

metrics:
  - name: Throughput
    filename: throughput
    query: flink_taskmanager_job_task_operator_componentThroughput
  - name: Memory Usage (MiB)
    filename: memory
    query: flink_taskmanager_Status_JVM_Memory_Heap_Used
  - name: CPU Load
    filename: cpu
    query: flink_taskmanager_Status_JVM_CPU_Load

workload:
  messagesPerSecond: 1
  experimentDuration: 3 # in seconds
  requestsPerMessage: 3
```

Figure 2: Example `global.yaml`

```yaml
- cpuTime: 5000    # in ms
  memoryUsage: 100 # in MiB
  outputSize: 1    # in KiB
- cpuTime: 5000
  memoryUsage: 200
  outputSize: 1
```

Figure 3: Example `components.yaml`, defining a list of components and their resource usage

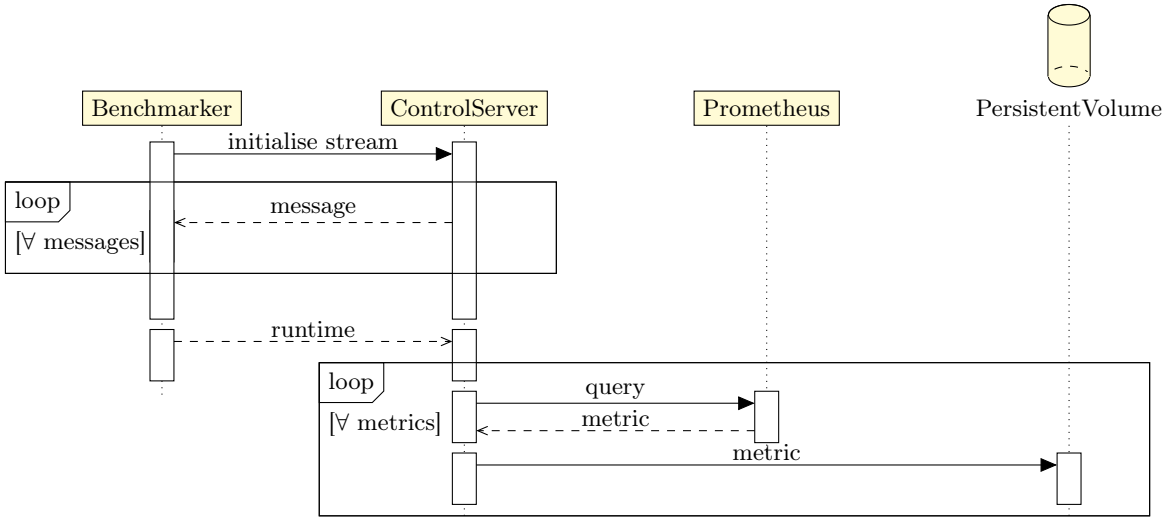


Figure 4: Communication between different parts of the system visualised as a UML sequence diagram

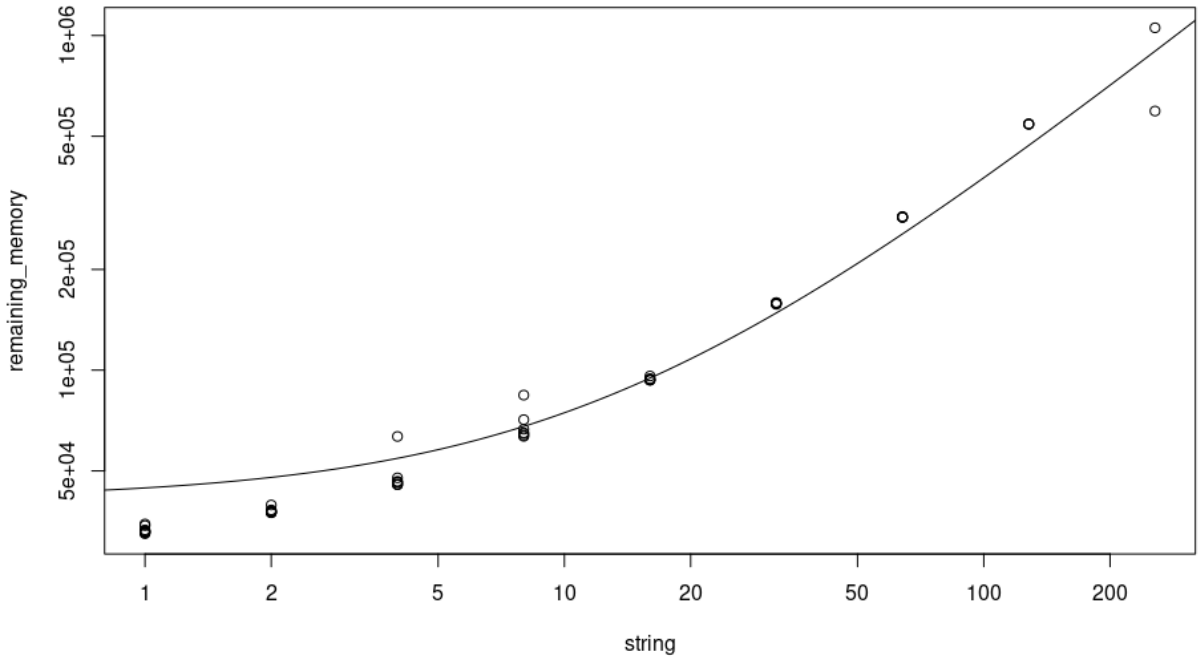


Figure 5: A log-log plot comparing the number of characters in a string and the observed memory unaccounted by the array. In each column, separate points correspond to different array sizes. The curve is a best-fit linear regression line.

We seek to know the average amount of memory used by a single character of a Java string. Knowing that a byte on an array takes up exactly one byte allows us to reformulate the problem to a simple linear regression shown in Figure 5. The model shows that overall memory usage can be expressed as

$$\text{memory usage} = 40\,\text{MiB} + \text{array size} + 3.268 \times \text{string size} + \epsilon, \tag{1}$$

contradicting the common wisdom that a character uses approximately two bytes of memory [7].

Figure 6 presents a more detailed view, but suggests the same conclusion. While the predictions seem to consistently overestimate memory consumption for short strings and similarly underestimate it for longer strings, adding a quadratic term is not enough to remove the bias in errors, and the errors are sufficiently small (see Section 3.1 for more details).

3.1 Adjusted Performance

We can use the two numerical parameters in Equation (1) to adjust our map class in order to ensure that it uses the correct amount of memory. We run a similar set of experiments as before, except replacing array size with expected memory usage as one of our independent variables (the other being string size). Memory usage is set to four different values: 64, 128, 256, and 512 MiB (note that the smallest possible memory usage is about 40 MiB), while string size is exponentially increased from 1 MiB up to the largest power of two small enough so that the string can be constructed from the array. We plot the errors in Figure 7. Note that the largest error is smaller than 50 KiB, which is good enough for our needs.

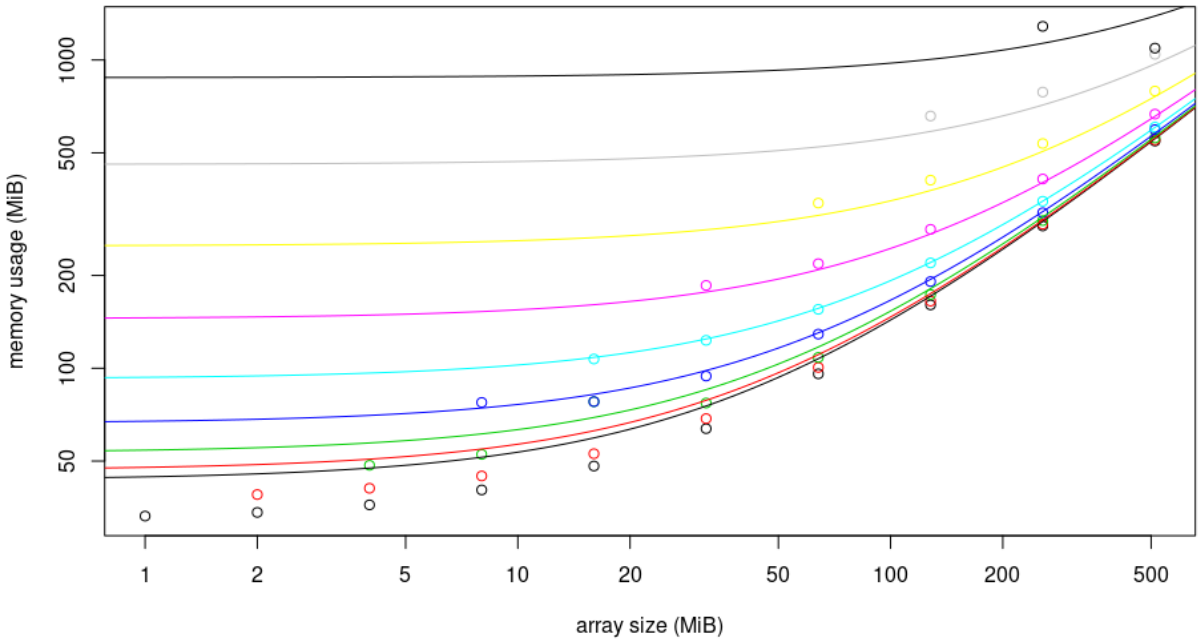


Figure 6: A log-log plot showing memory consumption across a range of array sizes, with different string sizes represented by different colours. For each string size, we also draw a regression line in the corresponding colour.

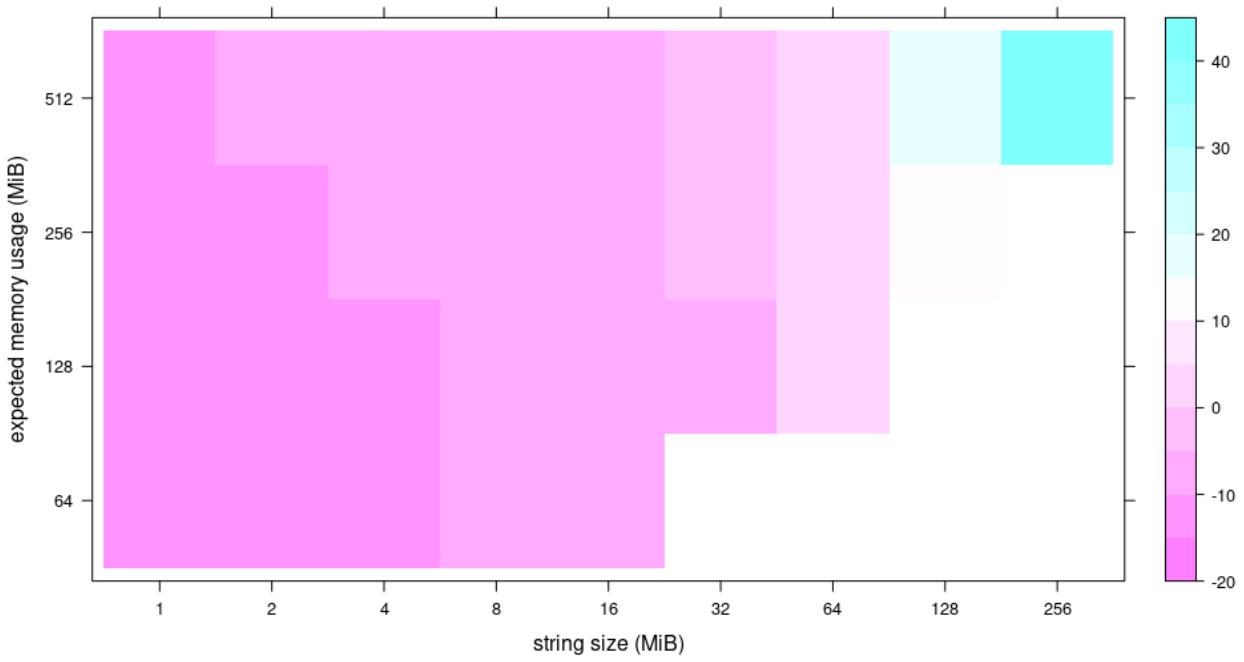


Figure 7: A heat map of errors (in KiB)

4 Experimental Evaluation

Experiments were performed in order to determine how well performance metrics observed with a standalone Java application transfer to MiniShift. We explore four values of `memoryUsage` (64 MiB, 128 MiB, 256 MiB, 512 MiB), while keeping `cpuTime` at zero so that each run lasts only as long as it takes to allocate and randomise the memory. For `outputSize`, we explore every power-of-two number of MiB compatible with the current `memoryUsage` value. We stick to a single component and record CPU and memory consumption at 1 s intervals using Prometheus. Each `memoryUsage` and `outputSize` configuration is written into `components.yaml` and run three times. With each run, we recreate all OpenShift components (pods, services, etc.), wait for the control server to terminate, and retrieve the generated JSON files.

Figure 8 shows CPU usage across all runs. Even though our standalone Java application easily reaches 100% CPU usage, when transferred to an OpenShift environment, a typical run could only get around 10%–15% (as indicated by the red curve), occasionally reaching up to 70% or 80% CPU usage. This can be explained by the fact that MiniShift internal processes as well as Flink JobManager and TaskManager are all running on the same machine. Even though the processes are distributed among eight cores, this overhead is sufficient to significantly decelerate the application.

Figure 9 shows similar memory usage measurements divided into four plots, one for each value of `memoryUsage`. We can see that there is significant variation among runs (and different `outputSize` values). In fact, in order to determine whether memory usage is optimal or hampered, one would need to run many identical experiments to account for variability. Moreover, each curve is unlikely to be fully summarised by a single number: maximum values are almost always higher than the expected result, while means are likely to be distorted by the initial several seconds of low memory usage as well as observed dips in memory usage later in the execution.

Note that individual runs can be summarised as follows:

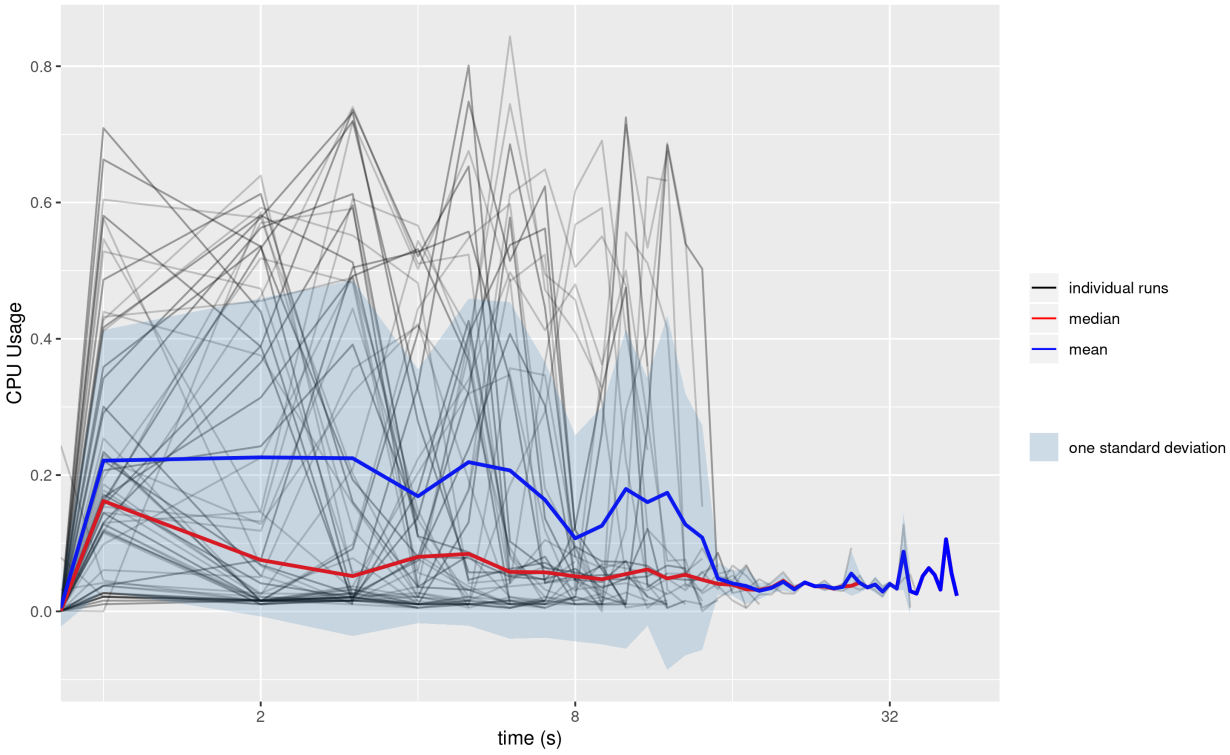
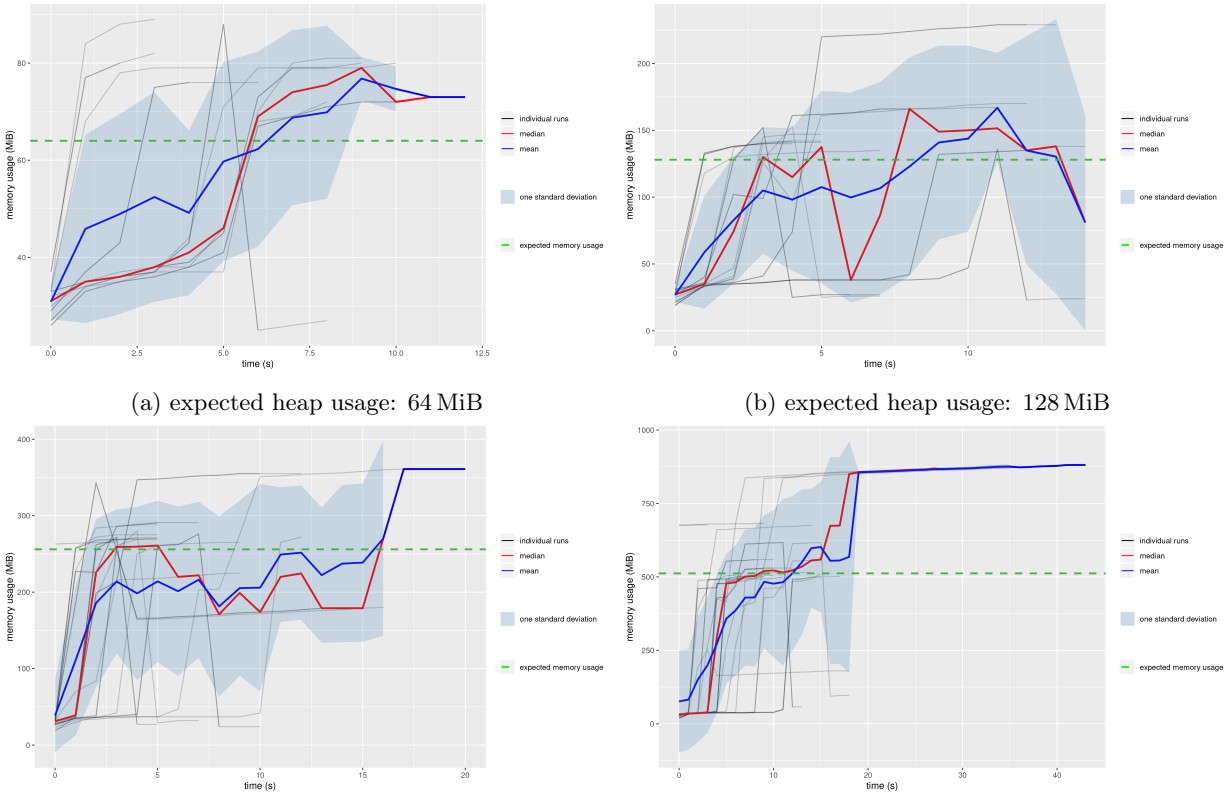


Figure 8: CPU usage across time. Each gray line represents a different run. The red curve is their (pointwise) median, the blue curve is the mean, while the shaded area marks one standard deviation around the mean. Note that time is on a log scale.



(a) expected heap usage: 64 MiB

(b) expected heap usage: 128 MiB

(c) expected heap usage: 256 MiB

(d) expected heap usage: 512 MiB

Figure 9: Observed versus expected memory usage across time for four expected memory amounts. The green dashed horizontal line marks the expected amount of memory usage. Each gray line represents a different run. The red curve is their (pointwise) median, the blue curve is the mean, while the shaded area marks one standard deviation around the mean.

1. Memory usage starts low.
2. It rises two times.
3. Sometimes memory usage experiences a significant drop, and sometimes this step is skipped.
4. Memory usage stays constant for a while.
5. The process terminates.

We can easily explain this pattern. The first increase is caused by the array allocation, while the second one is the result of constructing the output string. The drop in memory usage happens when the array is deallocated (garbage-collected) some time after the execution of my code completes. Sometimes that happens early enough to be captured by Prometheus, and sometimes the Flink job is marked as complete before garbage collection activates.

Finally, we consider the extent to which memory usage can be summarised by taking the maximum across time. We report each difference between expected E and observed O values as a relative error, i.e.,

$$\frac{O - E}{E}.$$

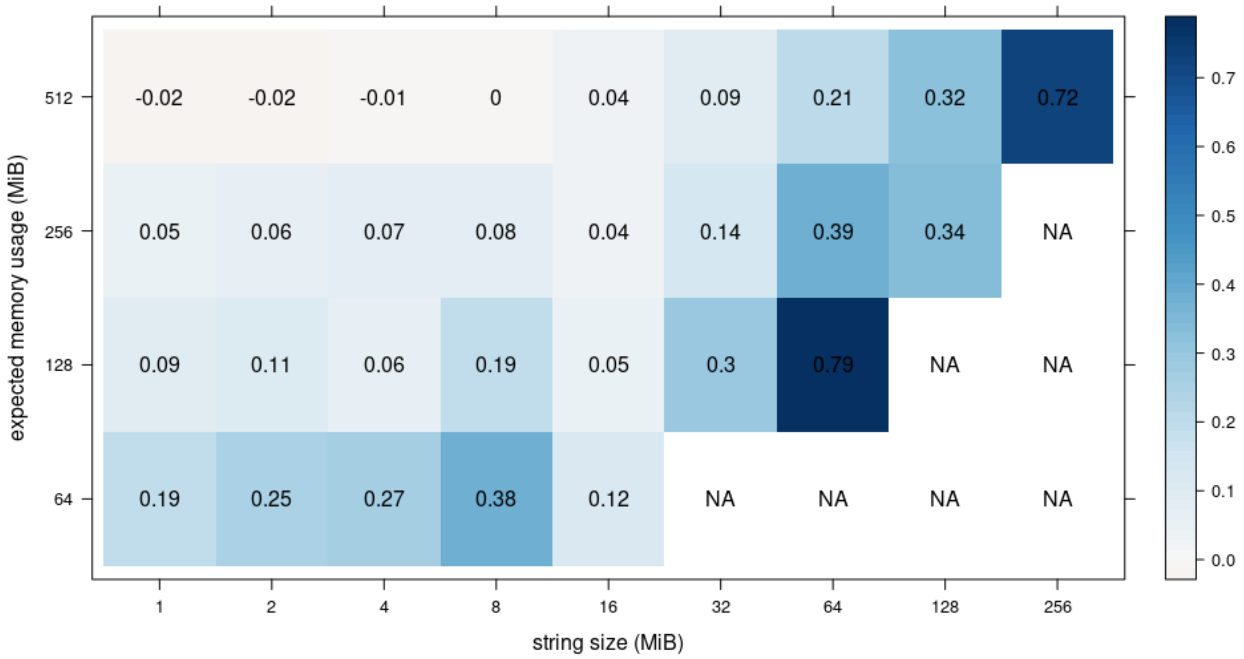


Figure 10: Relative median memory usage errors, as predicted by the maximum memory usage across execution

Table 1: Performance statistics of the top 100 e-commerce websites

Metric	min	mean	max
Page load time (s)	0.468	2.67	9.67
Page size (MiB)	0.719	3.03	14.21
Number of requests made per load	45	192	660

We consider these errors for all viable combinations of `memoryUsage` and `outputSize` and report the median of the three identical runs performed on each combination. The results are in Figure 10. Unsurprisingly, maximum memory usage across time is usually higher than the estimate. Also note that the overall shape of the heat map is similar to Figure 7, where we measure differences between observed and expected memory usage with the standalone Java application. In both cases, observed values are smaller with lower values of `outputSize`, and a combination of high overall memory usage and a long output string in the top right corner of both heat maps is likely to result in observed memory usage being significantly higher than the expected value. Even though we take median values to reduce the effect of outliers, observed memory usage can be up to 80% higher than the expected value, adding evidence to the imprecision and unreliability of making judgments based on a single number or a single experiment.

5 Example Applications

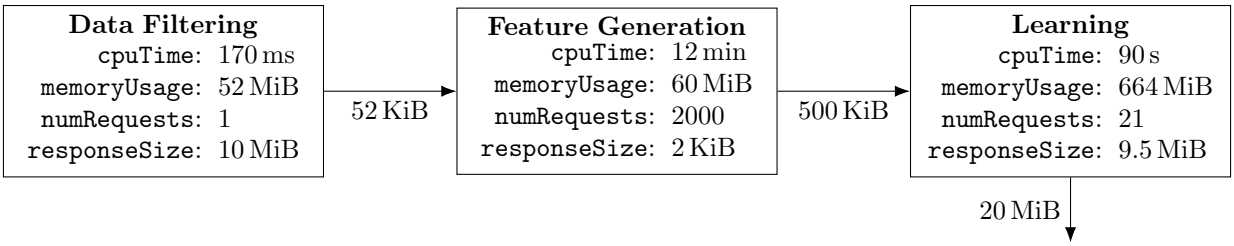


Figure 11: A model for a machine learning system, displaying typical resource usage metrics

A Simple Website In order to simulate a website, we need some data about the performance metrics of a typical website. We extract the data in Table 1 from experiments run on popular websites [6]. Thus, we can simulate an average website with a single component by making the following modelling assumptions:

- `cpuTime` $= \frac{\text{page load time}}{\text{number of requests per load}} = \frac{2.67\,\text{s}}{192} \approx 0.014\,\text{s}$;
- `memoryUsage` is minimal, i.e., the smallest amount necessary to construct the output string;
- `outputSize` $= \frac{\text{page size}}{\text{number of requests per load}} = \frac{3.03\,\text{MiB}}{192} = 16.16\,\text{KiB}$;
- `requestsPerMessage` = number of requests per load = 192.

A Machine Learning System For a realistic model of a machine learning (ML) system, we measured a classification system used to construct algorithm portfolios (akin to [4]) implemented using an R package `llama` [3] and run on a thousand maximum common subgraph problems and three algorithms. The resulting performance metrics are displayed in Figure 11, with each component's `outputSize` positioned next to its outgoing arrow. Sending a single message, then, corresponds to selecting a subset of data, generating features, and training an ML model. Note that the last two parameters define each component's I/O (i.e., file-reading) properties and can be easily ignored (otherwise, see Section 6 for more details). Other I/O-related parameters are assumed to be big enough (or small enough) so that their exact values are irrelevant to the performance of the system.

6 Input/Output Simulation

In order to simulate input/output (I/O) interactions similar to accessing a database or reading a file, we introduce a number of new component-specific parameters:

`ioMode` $\in \{\textbf{startup}, \textbf{regular}, \textbf{off}\}$ determines when to simulate I/O: during the initialisation stage, with every call to `map()`, or never.

`numRequests` is the number of request-response interactions between the component and the (simulated) data source.

`responseSize` is the size of the response (in KiB).

`databaseLatency` is the amount of time spent between a request and receiving the first byte of the response (in ms).

`bandwidth` is the bandwidth for transferring the response (as the request is assumed to be small) (in Mibit/s).

`intervalBetweenRequests` is the amount of time between receiving a response and sending another request (in ms).

We can then use these variables to simulate an I/O dialogue as described in Algorithm 1 (with unit conversion skipped for simplicity). We use the variable s to track our 'sleep debt', i.e., the amount of time the algorithm is supposed to sleep for at the end of each iteration on line 13. The function `mySleep` was added in order to ignore sleeping instructions for amounts of time smaller than 15 ms because most commonly-used operating systems have no support for this level of real-time behaviour control. We start by sleeping for `databaseLatency` ms to simulate the first delay between sending the request and receiving the data; other `databaseLatency` delays are incorporated into s on line 12. For each request, we gradually build up a linked list L until it uses about `responseSize` KiB memory. Setting the size of a single linked list node as `NODE_SIZE` allows us to calculate that we need $\frac{\texttt{responseSize}}{\texttt{NODE\_SIZE}}$ nodes in order to use the desired amount of memory (we round down in the algorithm). Similarly, in order to achieve the right `bandwidth`, we need each node to be constructed in $\frac{\texttt{NODE\_SIZE}}{\texttt{bandwidth}}$ time.

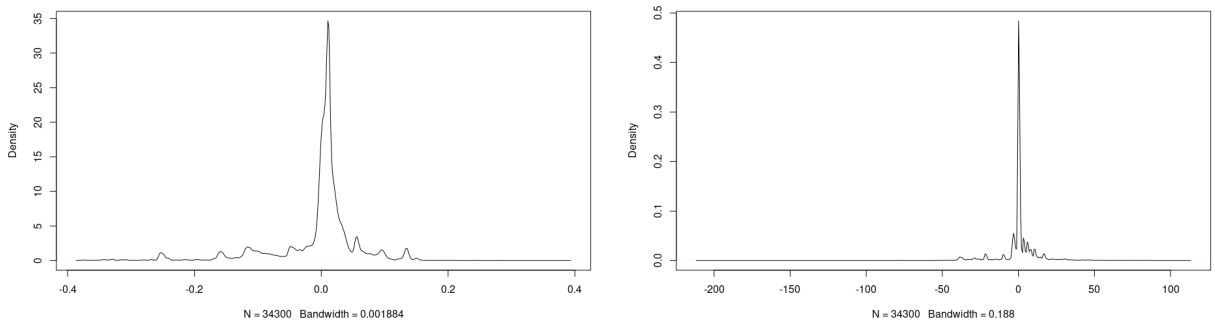
Based on separate experimental evidence, adding a single random integer to a linked list typically takes about 98 ns. While this amount of time is negligible for small lists, it can add up to significant delays when considering linked lists with millions of nodes. To take this into account, we measure the amount of time the yellow block of code takes to execute and subtract it from s on line 10. The yellow block is supposed to take $\frac{\texttt{responseSize}}{\texttt{bandwidth}}$ time, so we add the difference between the expectation and the reality to the amount of time the algorithm sleeps at the end of each iteration. If this is not the final iteration of the outer for loop, we also add `intervalbBetweenRequests` and `databaseLatency` as well. The function of s is to collect multiple reasons to sleep into a single call to `mySleep` as well as to take into account the amount of time it takes to build the linked list. If s is negative or too small, we let it influence the simulation of the next request so that the total amount of time is as accurate as possible. Otherwise, we sleep for the desired amount of time and reset s back to zero.

Algorithm 1: Simulation of a slow data transfer

```
1  s ← 0;
2  mySleep(databaseLatency);
3  for i ← 1 to numRequests do
4      t_0 ← time();
5      L ← new LinkedList of 64-bit integers;
6      for j ← 1 to ⌊responseSize / NODE_SIZE⌋ do
7          add a random integer to L;
8          mySleep(NODE_SIZE / bandwidth);
9      t_1 ← time();
10     s ← s + responseSize / bandwidth − t_1 + t_0;
11     if i < numRequests then
12         s ← s + intervalBetweenRequests + databaseLatency;
13     mySleep(s);
14     if s ≥ 15 ms then
15         s ← 0;
16 Function mySleep(t)
17     if t ≥ 15 ms then
18         sleep(t);
```

6.1 Is it using the right amount of memory?

Once again, we need to ensure that our program uses the right amount of memory (and to determine the value of `NODE_SIZE`). This time, we perform a much larger set of experiments where we measure memory



(a) relative errors

(b) errors (in MiB)

Figure 12: Density plots of errors

usage under a range of array, string, and linked list sizes:

$$\begin{aligned}\text{array size} &= 1, 3, 5, 7, 9, 10, 30, 50, 70, 90, 100, \ldots, 8 \times 10^8, 9 \times 10^8,\\ \text{string length} &= 1, 3, 5, 7, 9, 10, \ldots, 9 \times 10^6 \text{ (also ensuring that string length} \leq \text{array size)},\\ \text{number of nodes} &= 1, 3, 5, 7, 9, 10, \ldots, 9 \times 10^6.\end{aligned}$$

We also employ weighted simple linear regression in order to minimise squared relative error. This way, the model prioritises minimising error on smaller values, where the measured values are likely to be more accurate. More precisely, while the ordinary least squares approach would minimise

$$\sum_i (y_i - \hat{y}_i)^2,$$

we minimise

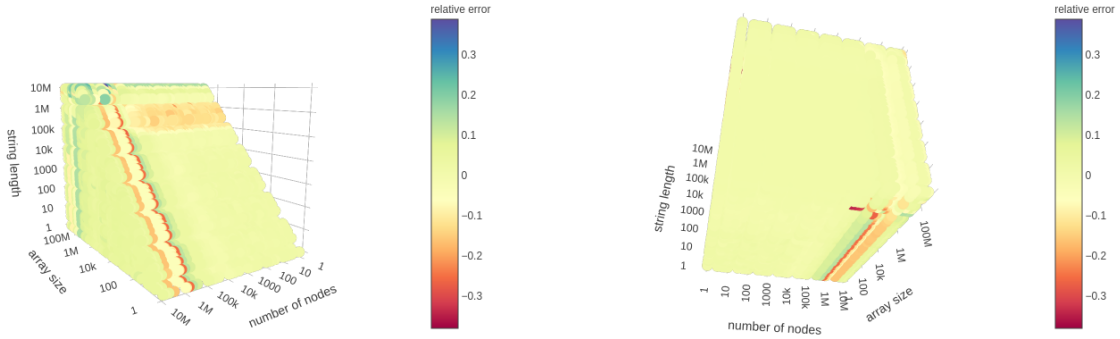
$$\sum_i \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2$$

instead. Here, y is the dependent variable (memory usage), y_i marks each recorded value, and $\hat{y}_i$ is the model's prediction of y_i according to the independent variables.

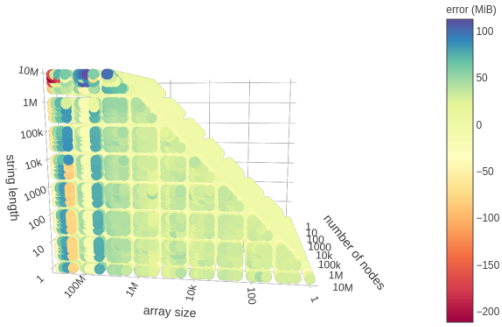
This model estimates base memory consumption to be $2.409 \times 10^7\,\mathrm{B} \approx 23\,\mathrm{MiB}$ and states that each element of a byte array uses 1.016 bytes, each character of a string uses 5.79 bytes, and each node of a linked list uses 61.94 bytes.

While relative errors vary from -38% to 39%, 88% of them are within $\pm 10\%$ (see density plots in Figure 12). Figure 13 clearly shows the number of nodes to be the troublesome variable: relative errors are at their worst with the number of nodes being around one million. Furthermore, as can be seen in Figure 13b, high (non-relative) errors only occur when the number of nodes parameter is at its highest. Using Figure 14 we can infer how the model functions under different values of the independent variables. To begin with, errors are stable throughout most of the string length values, except for the last few. For array size, we can see that the linearity assumption starts to break down towards the last one fourth of the values, where there is an increase in error magnitude, a shift towards primarily positive errors, and some significant negative outliers. The same story applies to the number of nodes as well, except there is a more significant drop in mean error per constant number of nodes, before the errors become primarily positive just like in Figure 14b.

Putting the estimates back into action We use the estimated memory usage parameters from local experiments to adjust the application so that it always uses a specified amount of memory. We use a smaller set of experiments to investigate the resulting application's accuracy. Namely, `memoryUsage` is set

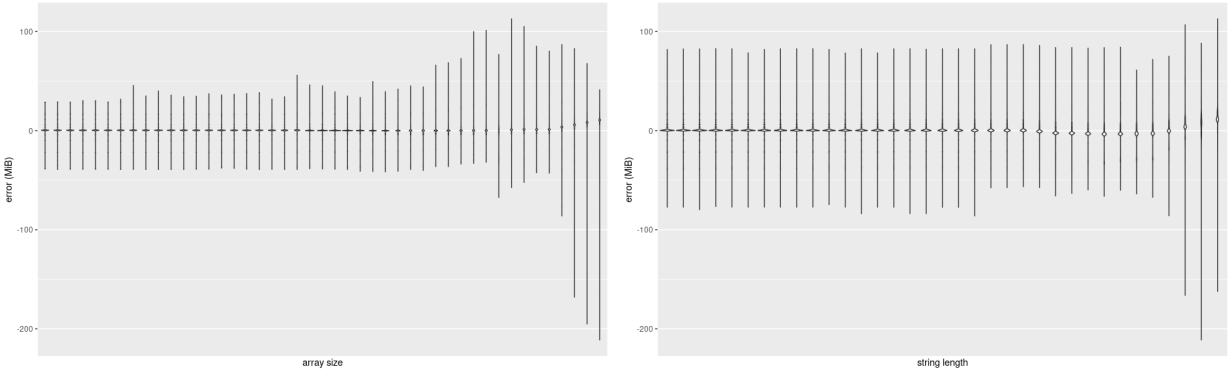


(a) relative errors



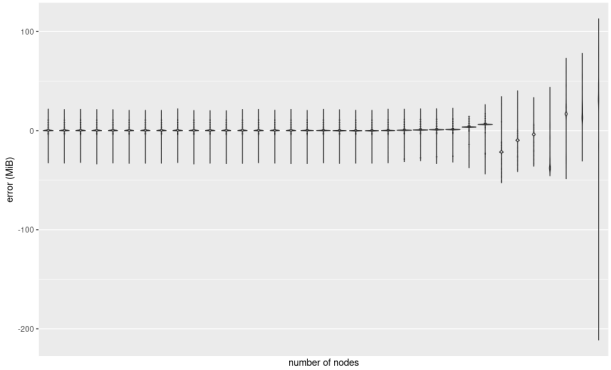
(b) errors

Figure 13: Prediction errors across all experiments



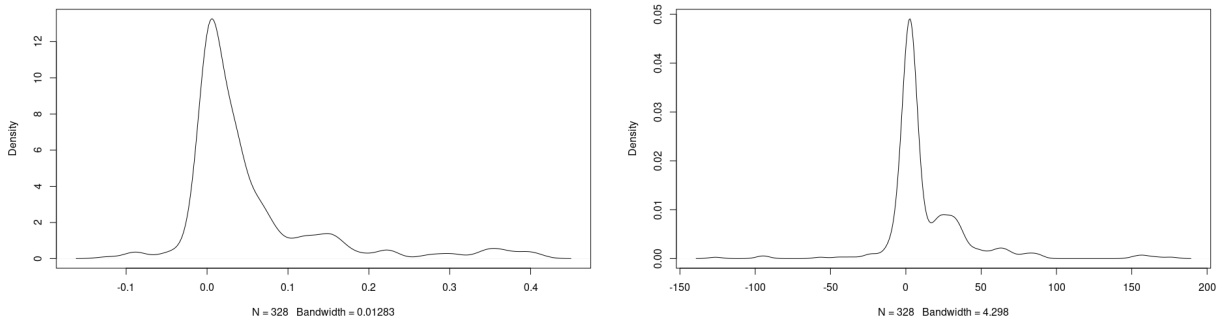
(a) array size

(b) string length



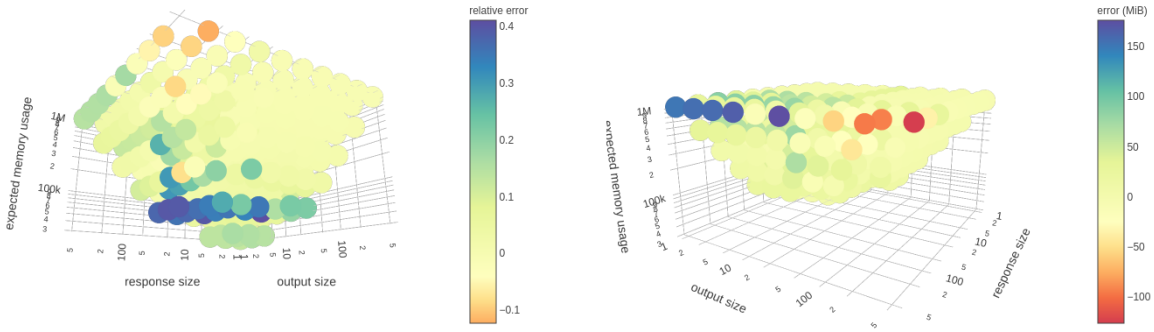
(c) number of nodes

Figure 14: Violin plots of errors against the three independent variables



(a) relative errors (b) errors (in MiB)

Figure 15: Density plots of errors



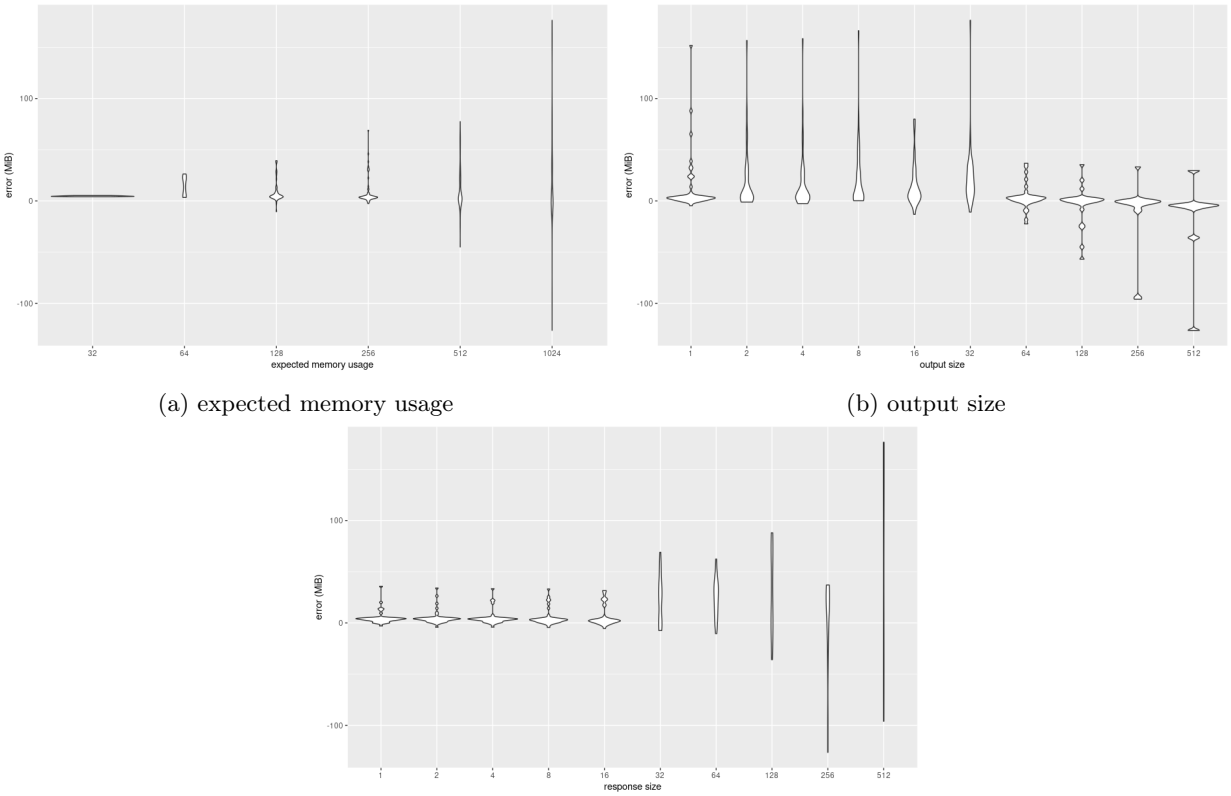
(a) relative errors (b) errors

Figure 16: Errors in memory usage, comparing the program's measured memory usage with the memory usage it was supposed to have

to $2^0, 2^1, 2^2, \ldots, 2^{10}$ MiB, while both `outputSize` and `responseSize` are set to any combination of powers-of-two MiB that satisfies total memory usage requirements. With each set of parameter values, the median of three runs is recorded.

While the errors have more variability, 82% of the relative errors are still within ±10% (see Figure 15). Unsurprisingly, highest relative errors occur with low (but not the lowest) expected memory usage (see Figure 16). Highest overall errors (both positive and negative) happen when `memoryUage` is at its highest, especially when `responseSize` is at its highest as well. This again alludes to the fact that a linear model is not a perfect fit for this situation and the growth of memory usage experiences non-linear effects with higher values.

Finally, the violin plots in Figure 17 show that errors tend to increase with higher `memoryUsage`. Our linear predictions seem to perform well with small linked lists, but get significantly less accurate with higher values of `responseSize`. Lastly, there is a positive-to-negative trend in errors with respect to increasing `outputSize` that is not captured by our model.



(a) expected memory usage

(b) output size

(c) response size

Figure 17: Violin plots of errors across independent variables

6.2 Is it taking the right amount of time?

If Algorithm 1 is working as intended, total running time of the I/O simulation process should be

$$\texttt{numRequests} \times \left(\texttt{databaseLatency} + \frac{\texttt{responseSize}}{\texttt{bandwidth}} \right) + (\texttt{numRequests} - 1) \times \texttt{intervalBetweenRequests}.$$

As the `sleep()` routine can be inaccurate for small arguments, we test a standalone implementation of the algorithm by recording its running time from within the program itself. We explore all combinations of the following values[5]:

$$\begin{aligned} \texttt{numRequests} &= 1, 2, 4, 8; \\ \texttt{responseSize} &= 0.001, 0.01, 0.1, 1, 10, 100, 1000\,\text{MiB}; \\ \texttt{databaseLatency} &= 0, 100, 1000\,\text{ms}; \\ \texttt{bandwidth} &= 0.001, 0.01, 0.1, 1, 10, 100, 1000\,\text{Mibit/s}; \\ \texttt{intervalBetweenRequests} &= 0, 100, 1000\,\text{ms}. \end{aligned}$$

Errors above 1 s only occurred with `responseSize` = `bandwidth` = 1000, i.e., when constructing a big linked list (with ~ 17 million nodes) with minimal `sleep()` breaks during the construction. In this case, the linked list construction itself takes more time than the entire process is supposed to take, with errors ranging from 0.8 s to 9.7 s. All other errors were within 0.5 s and most of them even within 0.002 s. As simulating slow memory buildup becomes less relevant when considering transfer speeds of around 1 Gibit/s or more, we conclude that the algorithm is sufficiently accurate.

7 Generating Various Usage Patterns

Hitherto the simulated user of the application was restricted to sending one or more messages at regular intervals. However, real usage patterns are often more complicated than that. For instance, a website might have more users during the day than at night, it might be slowly becoming more popular, and it might experience a spike in the number of users during an important event. In this section we present an approach that can simulate usage (or workload) patterns represented by any computable function.

Recall that a periodic workload is defined by the three parameters `experimentDuration`, `messagesPerSecond`, and `requestsPerMessage`. We define a 'functional' workload as a 4-tuple consisting of a `function` string and three floating-point numbers: `binWidth`, `initialX`, and `finalX`. The `function` variable contains a function of a single variable x expressed in JavaScript syntax. For example, in order to represent

$$f(x) = 10 + x + 5\sin(-4x + 4) + 10\exp\left(-\frac{(x-10)^2}{2}\right), \tag{2}$$

we would construct the following `function` string:

```
10 + x + 5 * Math.sin(-4 * x + 4) + 10 * Math.exp(-Math.pow(x - 10, 2) / 2).
```

Variables `initialX` and `finalX` define the interval of x values we want to simulate. Finally, `binWidth` is the discretisation parameter. In order to simulate the behaviour of a continuous function, we need to sample its values at different values of x, and `binWidth` defines the distance between different samples (see Figure 18).

The `function` defines the workload frequency, while x is the time variable. The number of messages we are supposed to send in an interval of size `binWidth` is then `binWidth` multiplied by the `function` evaluated at some point within the interval (in this case we choose the middle of the interval). We evaluate the `function` by replacing all occurrences of x with the value of x that corresponds to the interval of interest[6]

[5] Combinations of parameters resulting in high (above 1 min) expected running time were discarded in order to make the experimental process feasibly efficient.

[6] We only replace occurrences of x that are not part of a longer word. This is needed so that the `x` in `exp` is not replaced.

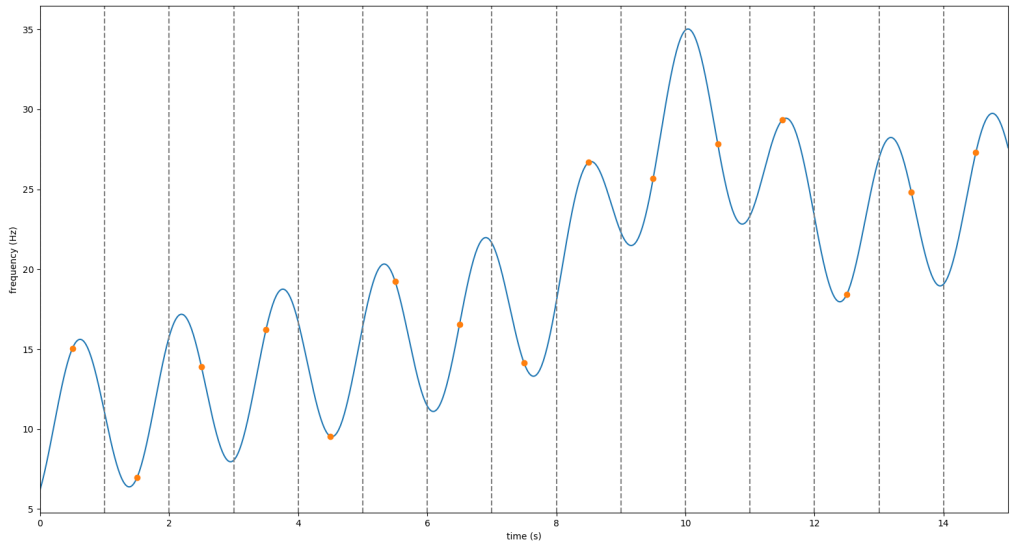


Figure 18: Function (2) plotted between $x = 0$ and $x = 15$ with intervals of `binWidth` marked by dashed vertical lines and the function's values at their centres marked by yellow dots

and sending the resulting string to a JavaScript evaluation engine (since Java itself has no evaluation function of its own). We then send the computed number of messages and wait `binWidth` time before evaluating the `function` at the next interval.

The implementation assumes that x (along with `binWidth`) is measured in seconds, and the `function` is measured in Hz. It is also important to set up the `binWidth` parameter so that the number of requests we send at a time is higher than zero, but also not too high—otherwise the simulation becomes inaccurate.

8 Validation

In this section we tackle the question: does the experimental data match our expectations? We have expectations for both memory and CPU usage. For memory, the parameter `memoryUsage` defines our expectation for how much memory should be used during the experiment (per component, that is). For CPU, we can measure how long one component takes to handle a single message and compare this data sample with `cpuTime`.

In both cases, we answer the question by fitting a probability distribution to the data and calculating the *p-value* of the expectation, i.e., the probability that a value generated by the fitted probability distribution is at least as far away from the mode[7] as our original expectation (see Figure 19 for an example). Note that in order for the fitted distribution to be realistic, we recommend running about twenty experiments for each configuration of parameters.

[7]We use modes instead of means for two reasons: • the mode represents a more reasonable best guess for our expectation (which is not an actual expected value of the distribution or the data!), • the distribution used for the CPU data could have an infinite mean for some values of its parameters.

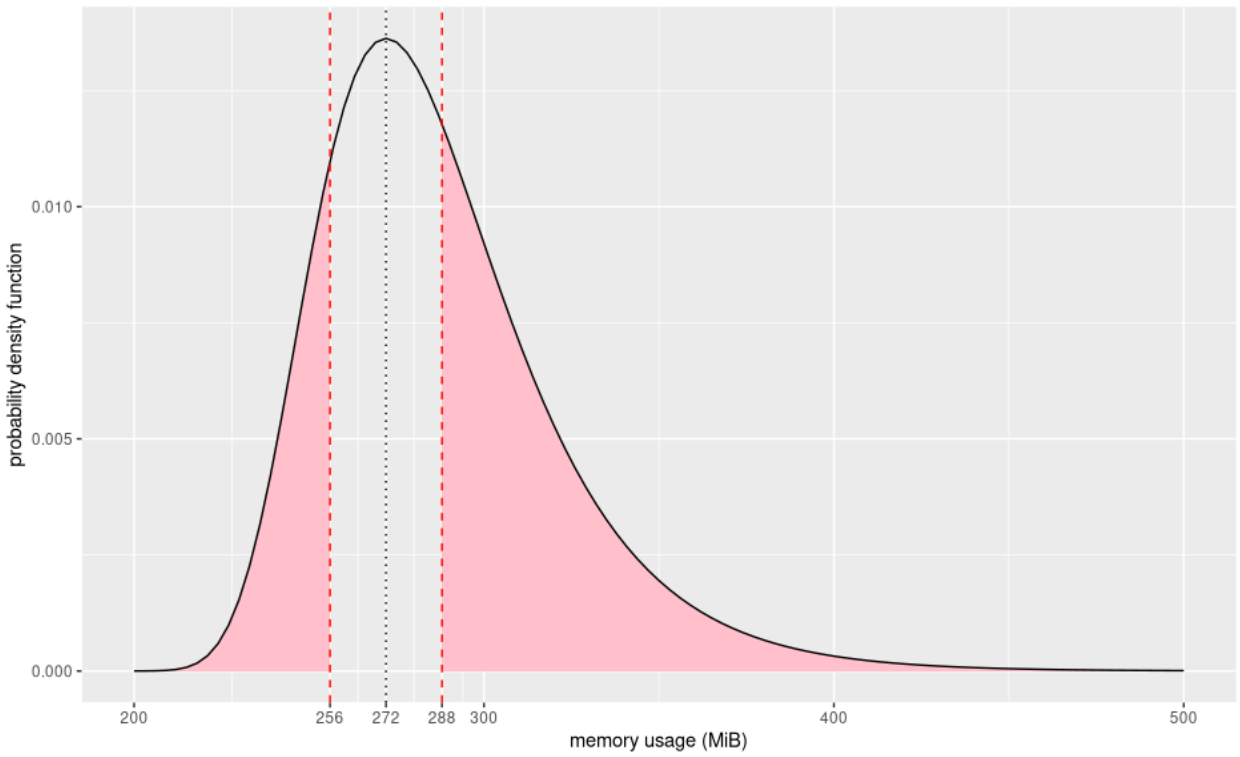


Figure 19: The visualisation of the p-value for memory usage data with `memoryUsage` set to 256 MiB. The p-value is calculated by integrating the shaded area under the probability density function. The black dotted line marks the mode of the fitted probability distribution. The first dashed red line shows our expected value of memory usage, while the second such red line is symmetric to the first one with respect to the mode of the distribution.

Table 2: Expected memory usage compared to the mode of the fitted distribution

Our expectation (MiB)	Mode (MiB)	p-value
64	77.1	0.056
128	128.1	0.998
256	272.3	0.583
512	514.9	0.983

8.1 Memory

Each experiment gives us a time series of memory usage values over time. We reduce each series to a single value by taking its maximum value, as lower values represent either memory usage before all variables are initialised or memory usage after some variables have already been claimed by the garbage collector. This collection of maximum values then follows Gumbel distribution, which is meant to represent any maximum values [1]. We fit this distribution to the data using the R package `ismev` [2] and calculate the p-values as previously described.

The results are summarised in Table 2. If we make an arbitrary decision to consider data as in line with our expectations if the p-value is greater than 0.05, then all four data sets pass this test. However, the p-value clearly shows that the 64 MiB experiment barely passes, while the 128 and 512 MiB experiments match our expectations extremely well!

The near-failure of the 64 MiB test can be partially explained by the fact that—unlike all other data sets—this one has no data points below the expected amount of memory usage (i.e., all measurements are above 70 MiB). This skews the data too far to the right. More generally, the goodness-of-fit plots in Figure 20 show a reasonable fit for all except perhaps the 128 MiB data set.

8.2 CPU Time

While we have CPU data expressed as a set of time series with values ranging between zero and one, our only CPU-based expectation is that of time, i.e., the amount of time it takes for one component to handle a single message. We can transform out data into this form by reducing each time series to the difference in time between first and last data points.

Without a solid statistical argument, we try fitting two asymmetric heavy-tailed distributions: log-normal and Fréchet. We are using a single data set with `cpuTime` set to zero. The plots in Figure 21 clearly show that the Fréchet distribution provides a much better fit than the log-normal distribution.

The mode of the Fréchet distribution with shape α, scale s, and location m can be calculated as

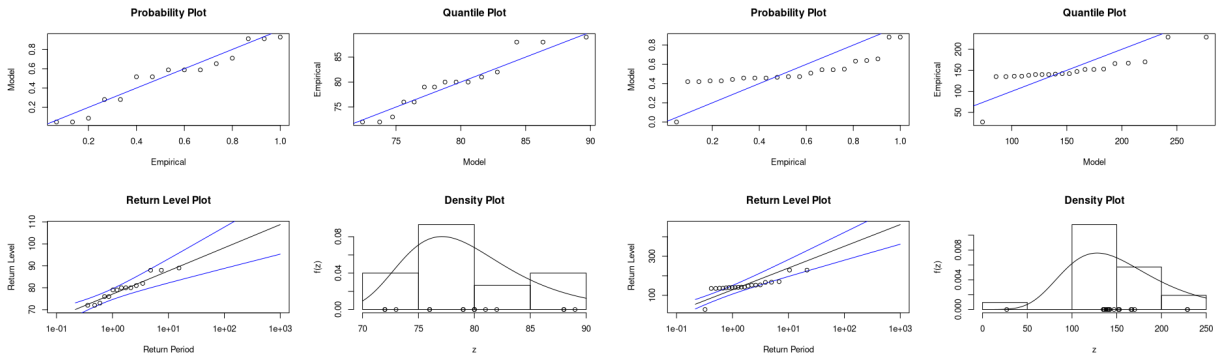
$$M = m + s\left(\frac{\alpha}{1+\alpha}\right)^{1/\alpha},$$

which with our data becomes $M = 6.1$. Recall that the expected amount of CPU time for this experiment is 0 s. Comparing it to the fitted distribution gives a p-value of 0.246—a value good enough to be accepted yet showing significant uncertainty stemming from the ambitious goal of having everything run in zero time.

9 Complex Component Topologies

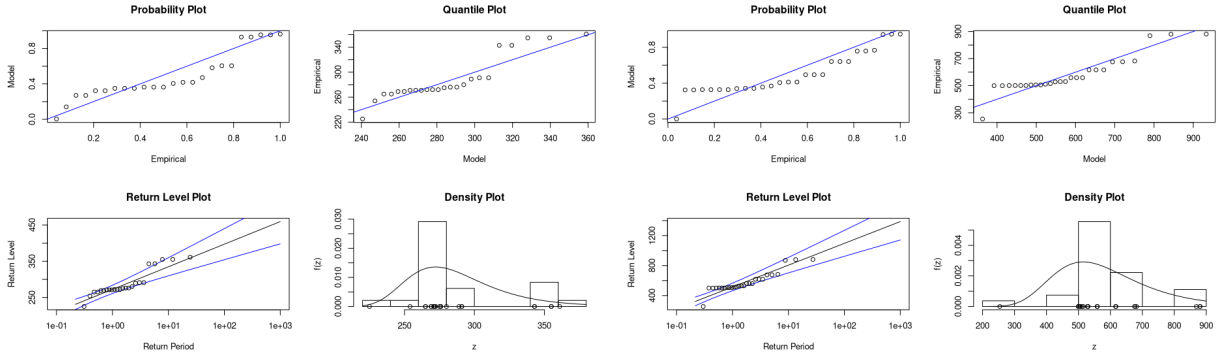
So far, our implementation only supports having a list of components where each component takes its input from its predecessor and produces output for its successor (see Figure 22a for an example). In this section, we introduce support for configurations of components expressed as a DAG.

An example component topology can be seen in Figure 22b. Here, the control server sends its messages to Components 1 and 2. After receiving a message, each component performs its work simulation (as



(a) expected memory usage: 64 MiB

(b) expected memory usage: 128 MiB



(c) expected memory usage: 256 MiB

(d) expected memory usage: 512 MiB

Figure 20: Goodness-of-fit plots for four different values of expected memory usage based on our experiments from Section 4

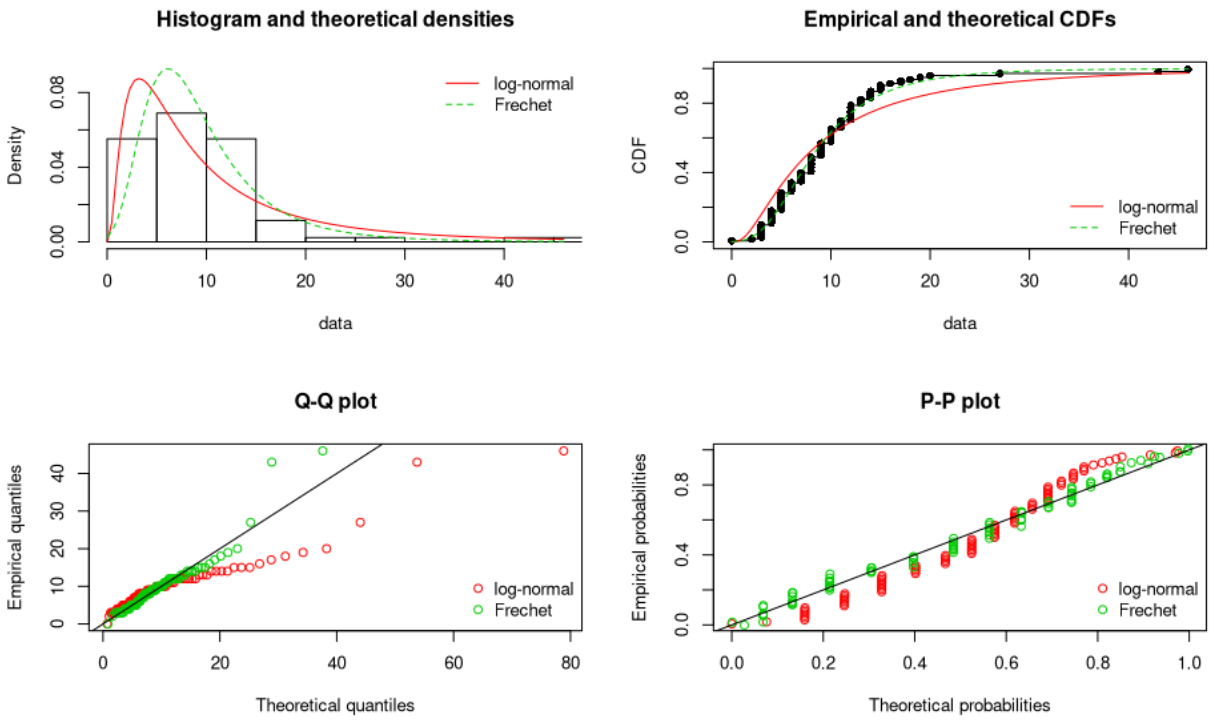
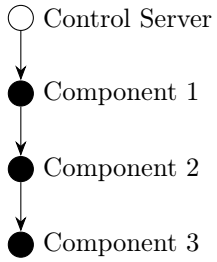
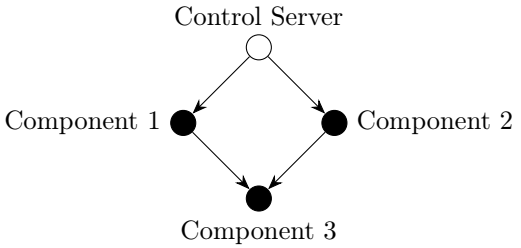


Figure 21: Evaluation of how well the log-normal and Fréchet distributions fit the data using four standard plots



(a) the topological structure imposed by the previous implementation



(b) an example of a DAG of components supported by the current implementation

```
- parents:
    - 0
  <...>
- parents:
    - 0
  <...>
- parents:
    - 1
    - 2
  <...>
```

(c) syntax for defining this DAG

Figure 22: The topology of components: before and now

defined in `components.yaml`) and forwards the message to Component 3. The experiment is complete once Component 3 finishes processing its last message.

In order to implement this, we extend the syntax of `components.yaml` to include an extra field for each component called `parents` representing a list of parent nodes in the graph structure. Figure 22c shows how our example DAG can be defined using this new field. The control server is always denoted by the number zero, while each component is assigned a number based on its place in the configuration file. Thus, the configuration file defines that Components 1 and 2 both have a single source of data, namely the control server, while Component 3 has Components 1 and 2 as its parents (i.e., sources, inputs, etc.). The rest of the configuration file is omitted for brevity. In order to avoid infinite cycles (and to make the implementation slightly simpler and nicer), we impose a restriction that Component i (i.e., the i'th component as defined in `components.yaml`) can only mention Component j as one of its `parents` if $j < i$. This design choice prohibits cycles and loops, while remaining flexible enough to define any DAG.

10 Conclusions and Future Work

- Measuring end-to-end latency using RabbitMQ
- Support more total memory usage and memory usage during I/O simulation using multiple arrays and linked lists

References

[1] E. J. Gumbel. Les valeurs extrêmes des distributions statistiques. In *Annales de l'institut Henri Poincaré*, volume 5, pages 115–158, 1935.

[2] J. E. Heffernan, A. G. Stephenson, and E. Gilleland. Ismev: an introduction to statistical modeling of extreme values. *R package version*, 1.42, May 2018.

[3] L. Kotthoff. LLAMA: leveraging learning to automatically manage algorithms. Technical Report arXiv:1306.1031, arXiv, June 2013.

[4] L. Kotthoff, C. McCreesh, and C. Solnon. Portfolios of subgraph isomorphism algorithms. In P. Festa, M. Sellmann, and J. Vanschoren, editors, *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2016.

[5] J. J. O'Connor and E. F. Robertson. Lothar Collatz. `http://www-history.mcs.st-andrews.ac.uk/Biographies/Collatz.html`, November 2006.

[6] SolarWinds. Web performance of the world's top 100 e-commerce sites in 2018. https://royal.pingdom.com/web-performance-top-100-e-commerce-sites-in-2018/, March 2018.

[7] M. Vorontsov. An overview of memory saving techniques in Java. http://java-performance.info/overview-of-memory-saving-techniques-java/, June 2013.