Automated Benchmarking of Container Applications

Paulius Dilkas

31st July 2019

1 Introduction

Before an application is deployed to a cloud, one needs to select the required number of resources. But what is the right resource configuration for *my* application? One might ask and wonder whether there is a way to obtain a clear answer.

We present a benchmarking system that can simulate the workflow of a range of distributed applications, measure various performance metrics, and detect whether a given resource configuration is sufficient for the application to perform well.

The system comes with a number of 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 to represent, e.g., additional components of the application.

Our simulated applications consist of a sequence of maps implemented using Apache Flink. They receive messages from a *control server* and perform a predefined amount of work, using a predefined amount of memory. The server can send any number messages at specified intervals (or in more interesting patterns, if I get around to implementing that).

The Flink application is enclosed in a Docker container and deployed on the OpenShift cloud platform. Performance metrics are recorded using Prometheus, which is assumed to be a separate application deployed on the same OpenShift platform.

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 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 two simple Dockerfiles.

The first one is used for both TaskManagers and the JobManager and extends the original Flink Docker image by enabling Prometheus support on port 9250. Prometheus can then use that port to retrieve and record performance metrics.

The second Dockerfile extends the first one by adding a JAR file with Java code for both the control server and the Flink app. This image also contains a custom `ENTRYPOINT` shell script that sends the Flink app to the JobManager (via port 8081) as a background process, while executing the control server in the foreground. This is the optimal arrangement of the two processes since the control server always waits for the Flink app to finish in order to take its running time into account when requesting data from Prometheus.

In both Dockerfiles, one needs to change the permissions and group ownership of the `/opt/flink` directory (to `775` and `root` respectively) so that the containers can be successfully executed by any user belonging to the group `root`. Both images were put on Docker Hub in order to make them easily accessible by MiniShift.

A Docker Compose file can then be used to combine several Dockerfiles into a valid deployment configuration. In this file we define three services: `Control`, `JobManager`, and `TaskManager`, establishing open ports as pictured in Figure 1. This file was then converted to OpenShift manifestos using Kompose[1]. 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 two 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 basic networking information along with three parameters that control the experiment as well as a list of metrics. The experiment-specific parameters control how often to send messages (`messagesPerSecond`), how long the experiment should last (`experimentDuration`), and how many messages to send at a time (`requestsPerMessage`). Each metric is described with three properties: `name`, `filename`, and `query`. The last one corresponds to the name of the property as defined by Prometheus, while the other two are used for data storage and plotting. 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[2] had to be modified in order to disable OAuth-based authentication by replacing

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

This may or may not be a problem when moving from MiniShift to an actual cloud. 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 a chain of mappers 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]http://kompose.io/
[2]https://github.com/minishift/minishift-addons/tree/master/add-ons/prometheus

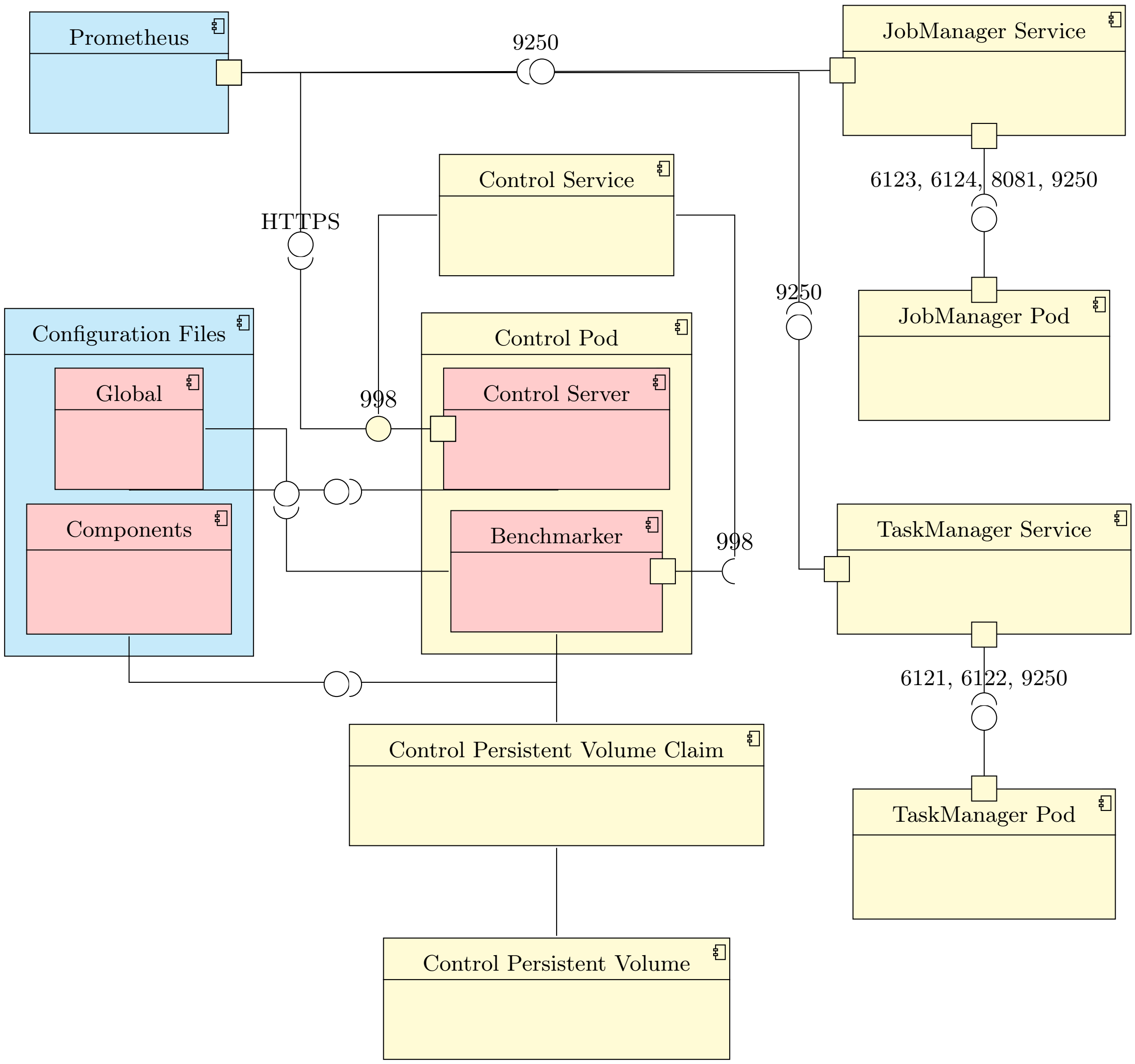


Figure 1: UML component diagram of the system, as deployed on MiniShift. Yellow components are OpenShift manifestos, while red components represent files (either Java classes or YAML configuration files). Network connections are shown with ports and have port numbers (or application-layer protocol names) displayed.

```yaml
controlHostname: control
controlPort: 9998
prometheusHostname: prometheus
messagesPerSecond: 1
experimentDuration: 3 # in seconds
requestsPerMessage: 3
metrics:
  - name: Throughput
    filename: throughput
    query: flink_taskmanager_job_task_operator_componentThroughput
  - name: Heap Usage (MiB)
    filename: heap
    query: flink_taskmanager_Status_JVM_Memory_Heap_Used
  - name: CPU Load
    filename: cpu
    query: flink_taskmanager_Status_JVM_CPU_Load
```

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 requirements

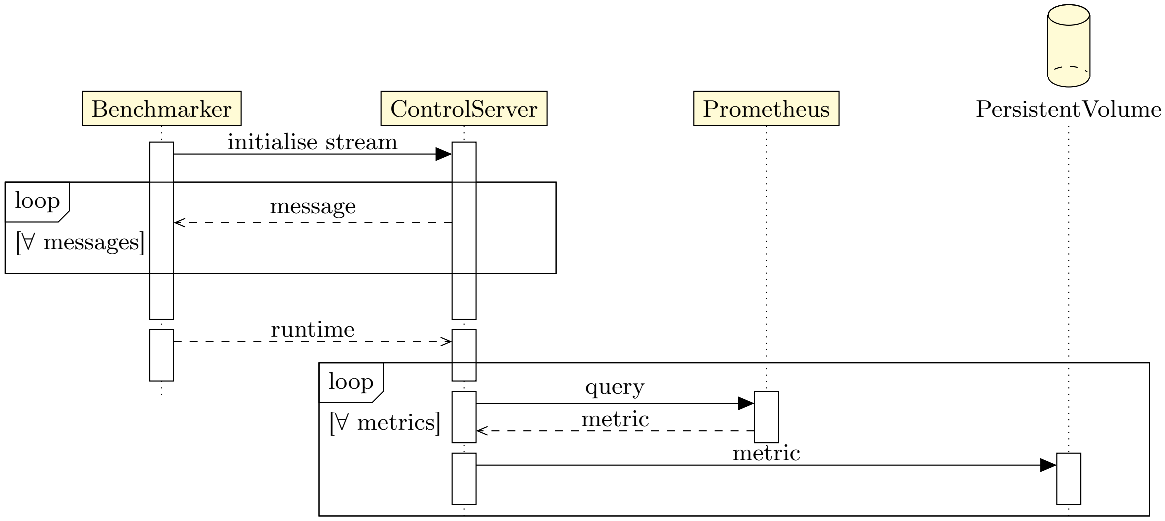


Figure 4: Communication between different parts of the system visualised as a UML sequence diagram

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 [3] 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 $2^0, 2^1, 2^2, \dots, 2^9$ and output strings of $2^0, 2^1, 2^2, \dots, \min\{2^8, \text{array size}\}$ 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[3] and its Maximum Resident Set Size metric. Each experiment was repeated three times, and median values were taken.

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

[3] `https://www.gnu.org/software/time/`

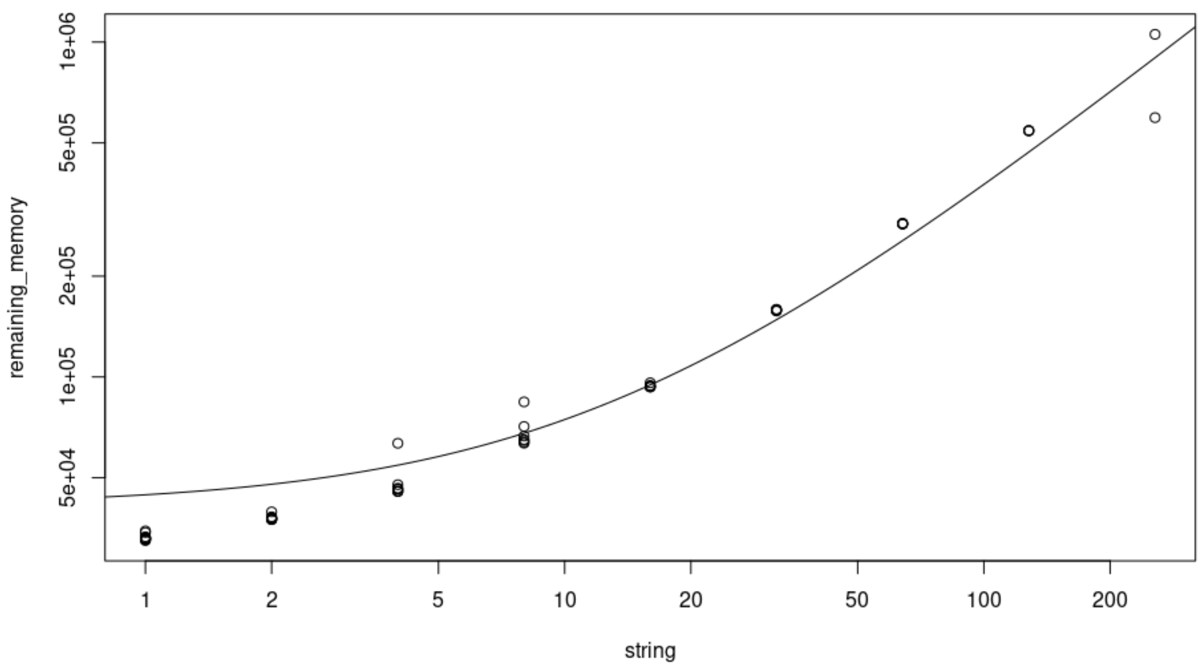


Figure 5: A log-log plot comparing the number of characters in a string and the observed memory unaccounted by the array. In each column, separate points correspond to different array sizes. The curve is a best-fit linear regression line.

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 [5].

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.

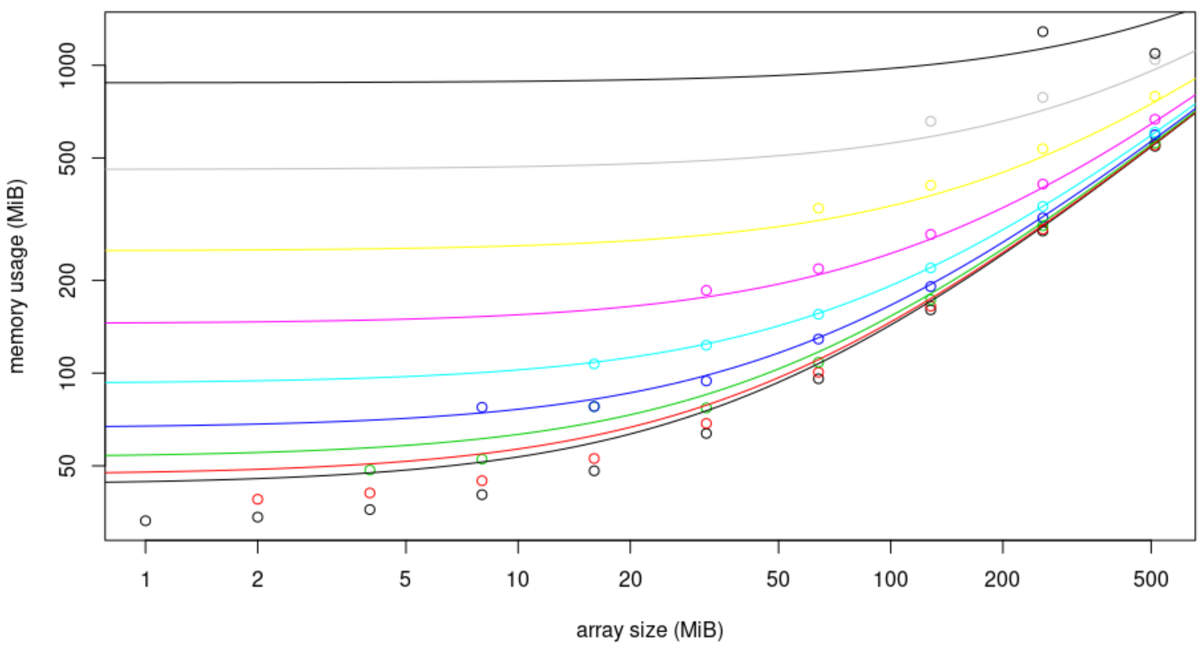


Figure 6: A log-log plot showing memory consumption across a range of array sizes, with different string sizes represented by different colours. For each string size, we also draw a regression line in the corresponding colour.

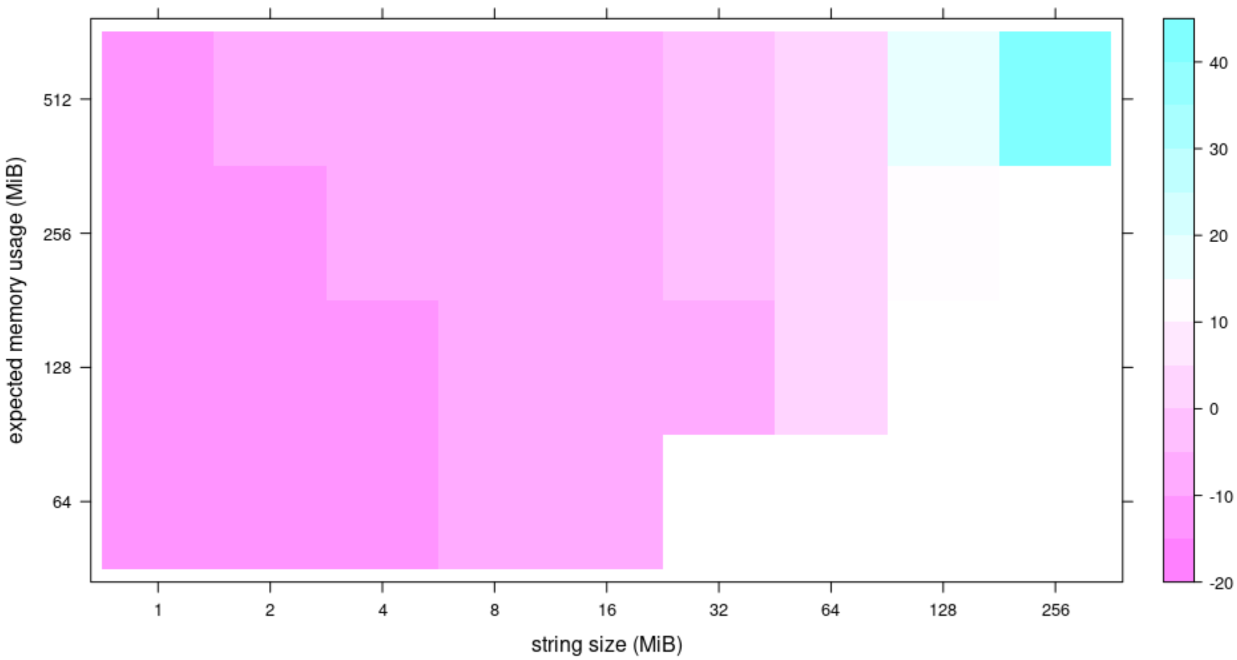


Figure 7: A heat map of errors (in KiB)

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:

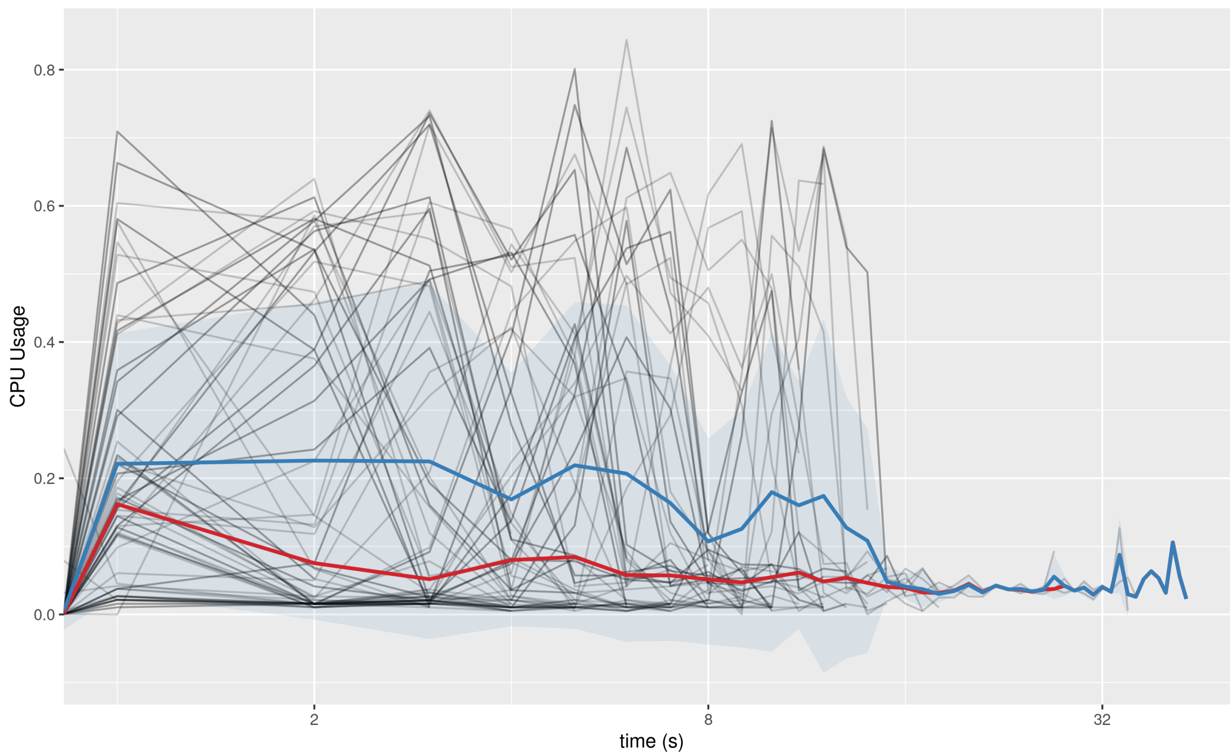
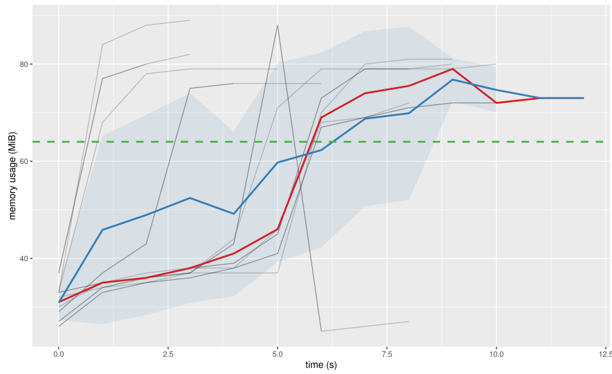
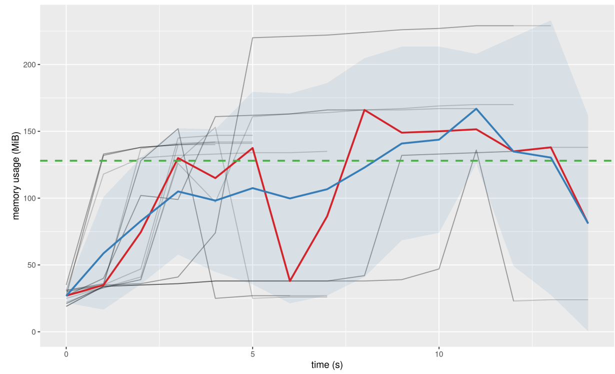


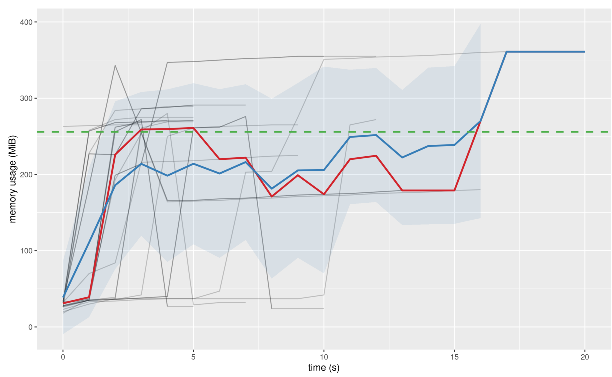
Figure 8: CPU usage across time. Each gray line represents a different run. The red curve is their (pointwise) median, the blue curve is the mean, while the shaded area marks one standard deviation around the mean. Note that time is on a log scale.



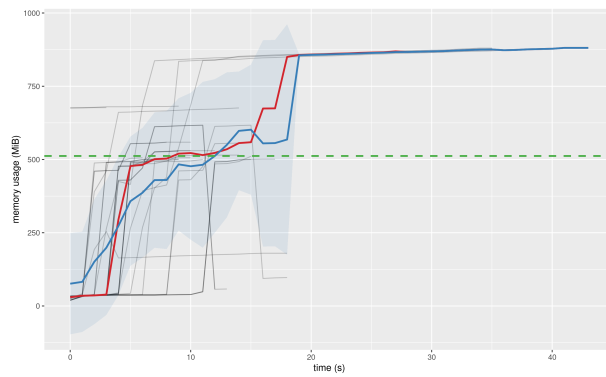
(a) expected heap usage: 64 MiB



(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.,

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

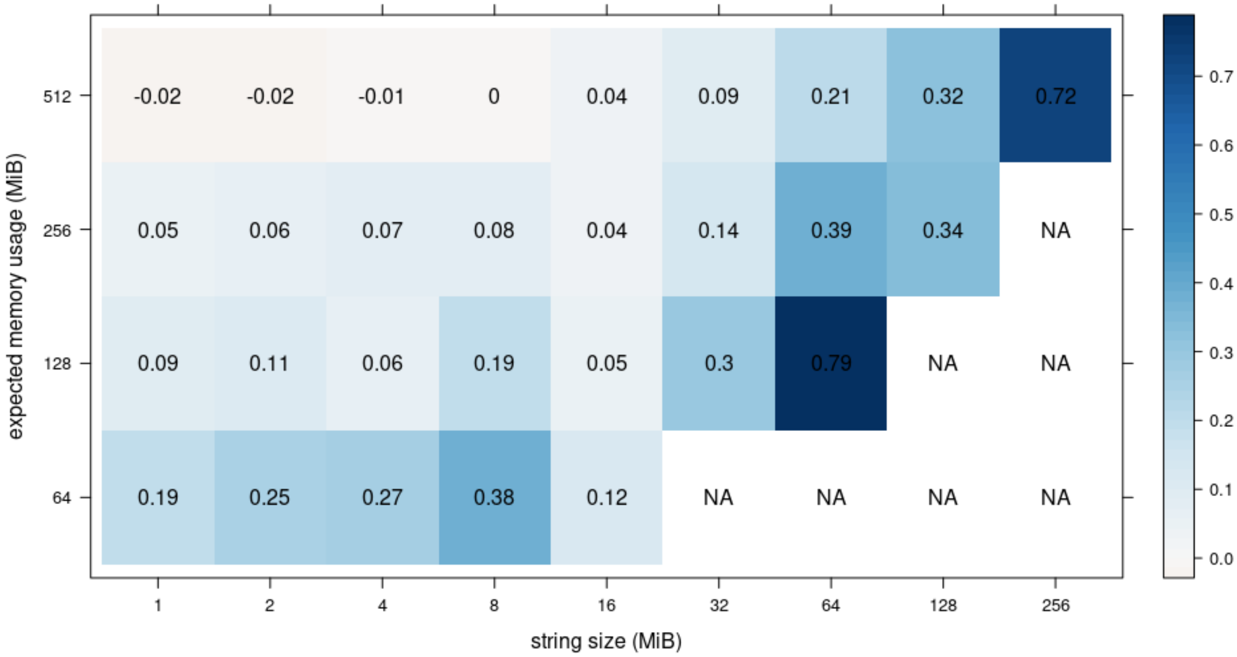


Figure 10: Relative median memory usage errors, as predicted by the maximum memory usage across execution

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

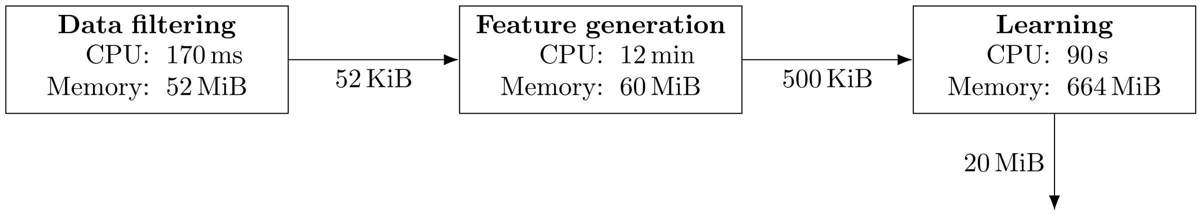


Figure 11: A model for a machine learning system, displaying typical resource usage metrics

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 [4]. 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 [2]) implemented using an R package Llama [1] and run on a thousand maximum common subgraph problems and three algorithms. The resulting performance metrics are displayed in Figure 11. Sending a single message, then, corresponds to selecting a subset of data, generating features, and training an ML model.

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:

`databaseOnStartup` = **true** means that I/O will be simulated only once, during the initialisation stage. Otherwise it will be simulated with every call to `map()`.

`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 B/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 and rounding operations skipped for simplicity). We use a linked list L to gradually construct an object of size `responseSize`, simulating a big file slowly being uploaded to 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 simulate a file transfer of size `responseSize`. Similarly, in order to achieve the right `bandwidth`, we need each node to be constructed in $\frac{\texttt{NODE\_SIZE}}{\texttt{bandwidth}}$ time (we are assuming that adding a random integer to a list takes a negligible amount of time).

Algorithm 1: Simulation of a slow network data transfer

```
for i ← 1 to numRequests do
    sleep(databaseLatency);
    L ← new LinkedList of 64-bit integers;
    for j ← 1 to responseSize/NODE_SIZE do
        add a random integer to L;
        sleep(NODE_SIZE/bandwidth);
    if i < numRequests then
        sleep(intervalBetweenRequests);
```

6.1 Is it using the right amount of memory?

Once again, we need to calibrate the problem to ensure that it uses the right amount of memory (and also to determine the value of `NODE_SIZE`). This time, we perform a much larger set of experiments, where the independent variables are assigned the following values:

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

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

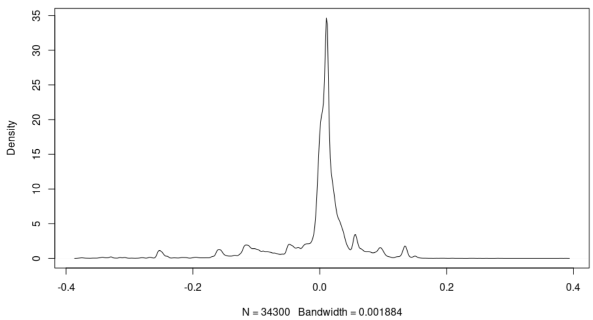
$$\sum_i \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2$$

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.

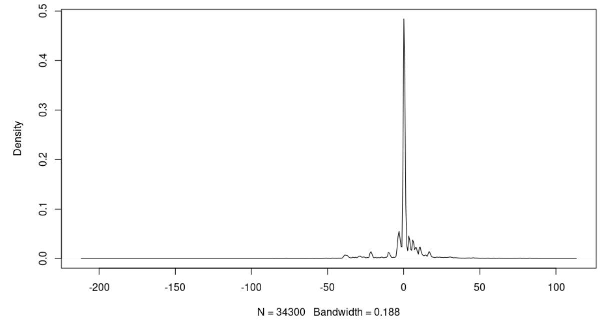
This model estimates base memory consumption to be $2.409 \times 10^7\,\text{B} \approx 23\,\text{MiB}$ and states that each element of a byte array uses 1.016 bytes, each character of a string uses 5.79 bytes, and each node of a linked list uses 61.94 bytes.

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.

Results 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 to $2^0, 2^1, 2^2, \ldots, 2^{10}$ MiB, while both

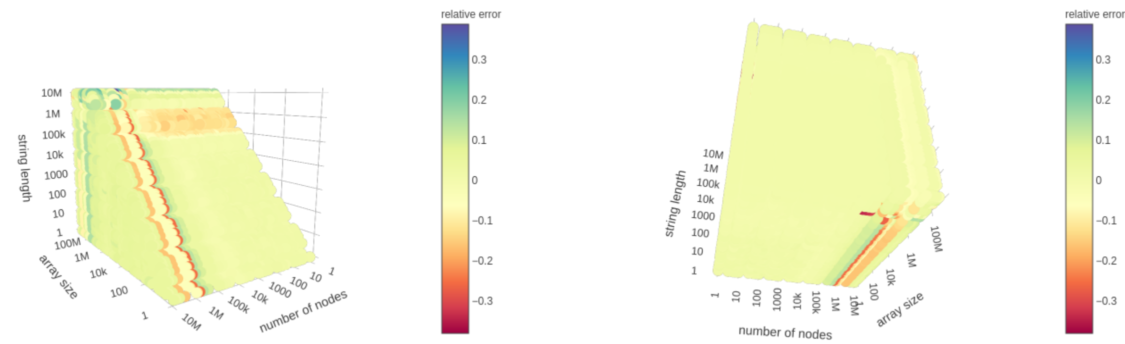


(a) relative errors

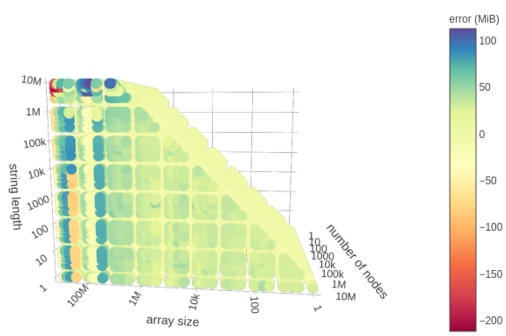


(b) errors (in MiB)

Figure 12: Density plots of errors

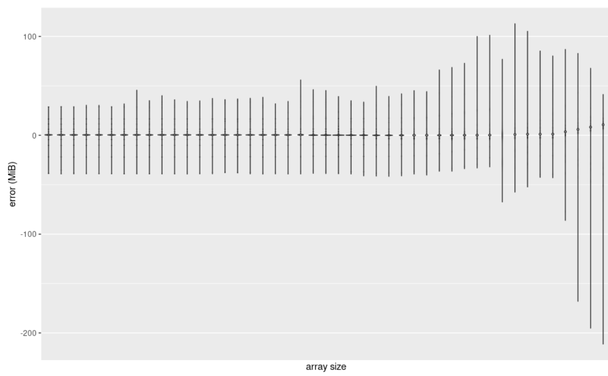


(a) relative errors

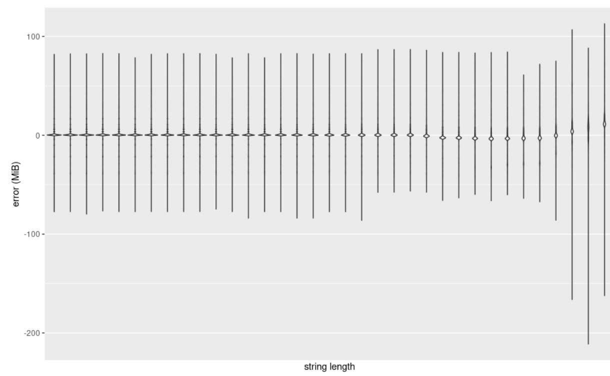


(b) errors

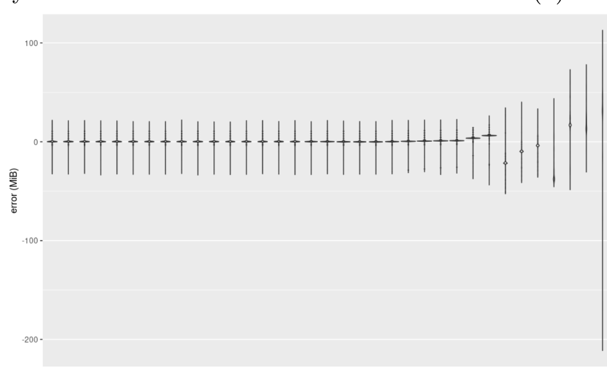
Figure 13: Prediction errors across all experiments



(a) array size

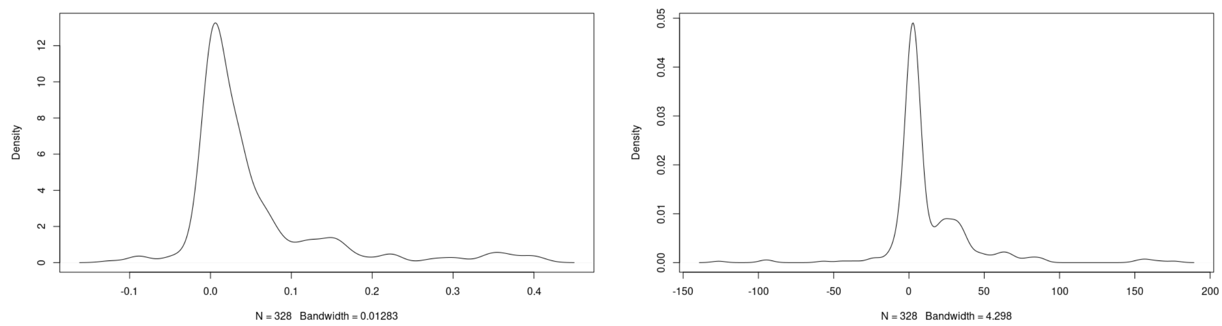


(b) string length



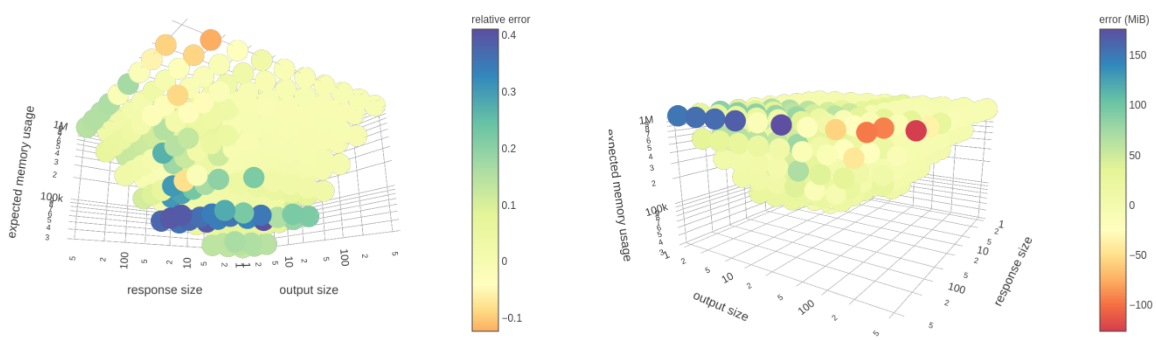
(c) number of nodes

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

`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.

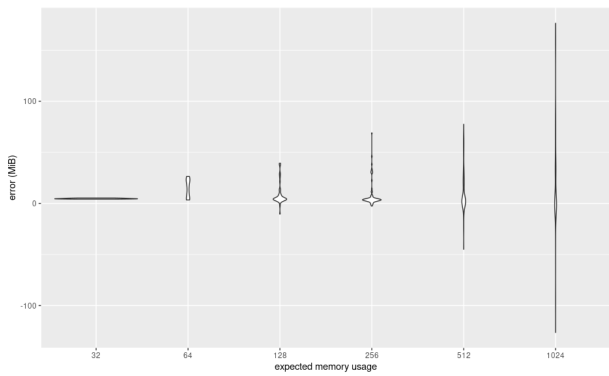
While the errors have more variability, 82% of the relative errors are still within ±10% (see Figure 15). In terms of relative errors, most of the problematic values are at 64 MiB `memoryUsage` (see Figure 16). Highest overall errors (both positive and negative) happen when `memoryUage` is at its highest, especially when `responseSize` is at its highest as well. This again alludes to the fact that a linear model is not perfect 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 (unsurprisingly) 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 that is not captured by out model, achieved by increasing `outputSize`.

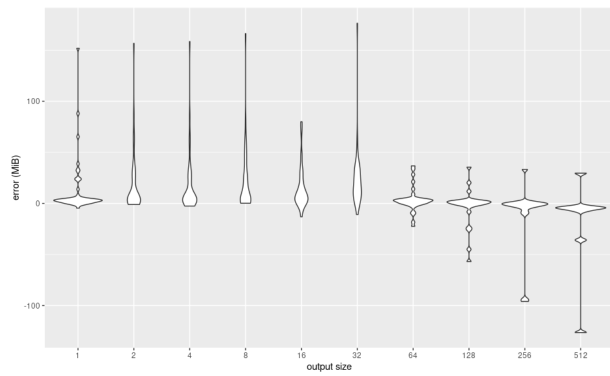
6.2 Is it taking the right amount of time?

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

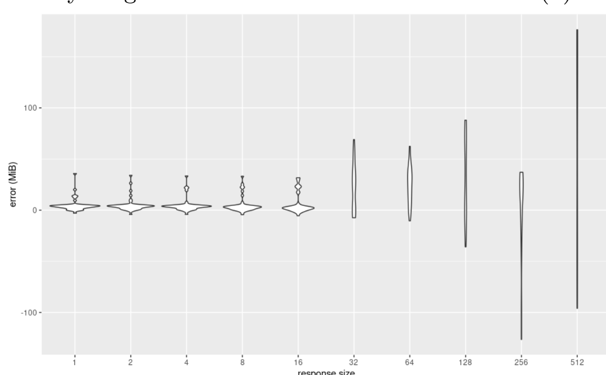
$$\texttt{numRequests} \times \left(\texttt{databaseLatency} + \frac{\texttt{responseSize}}{\texttt{bandwidth}}\right) + (\texttt{numRequests} - 1) \times \texttt{intervalBetweenRequests}.$$



(a) expected memory usage



(b) output size



(c) response size

Figure 17: Violin plots of errors across independent variables

References

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

[2] 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.

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

[4] 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.

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