Modeling Real-World Load Patterns for Benchmarking in Clouds and Clusters

Kashifuddin Qazi
Department of Computer Science
Manhattan College
NY, USA

***Abstract*—Cloud computing has currently permeated all walks of life. It has proven extremely useful for organizations and individual users to save costs by leasing compute resources that they need. This has led to an exponential growth in cloud computing based research and development. A substantial number of frameworks, approaches and techniques are being proposed to enhance various aspects of clouds, and add new features. One of the constant concerns in this scenario is creating a testbed that successfully reflects a real-world cloud datacenter. It is vital to simulate realistic, repeatable, standardized CPU and memory workloads to compare and evaluate the impact of the different approaches in a cloud environment. This paper introduces Cloudy, which is an open-source workload generator that can be used within cloud instances, Virtual Machines (VM), containers, or local hosts. Cloudy utilizes resource usage traces of machines from Google and Alibaba clusters to simulate up to 16000 different, real-world CPU and memory load patterns. The tool also provides a variety of machine metrics for each run, that can be used to evaluate and compare the performance of the VM, container or host. Additionally, it includes a web-based visualization component that offers a number of real-time statistics, as well as overall statistics of the workload such as seasonal trends, and autocorrelation. These statistics can be used to further analyze the real-world traces, and enhance the understanding of workloads in the cloud.**

***Keywords*—Cloud computing; workload generator; cluster computing**

I. Introduction

Cloud computing has become an important part of most organizations these days. By leasing compute resources from cloud providers, organizations can save on hardware and setup costs. Because of its popularity cloud computing has garnered substantial attention within the research community. Research is being consistently performed on a large scale on the cloud to make it faster, more efficient, and add more features.

Researchers have explored different aspects of cloud computing such as live migration [1] [2], vertical elasticity [3], horizontal elasticity, remote memory [4], workload prediction, container placement, virtual machine consolidation, and load balancing [5]. In order to evaluate proposed approaches similar to these, it is important to have a standard tool that can be used to benchmark them. This entails an environment that simulates the resource usage patterns seen in the real world. For example, it is important to test live migration approaches with virtual machines using up realistic amounts of memory over time. The environment should also offer a number of features to be useful as an evaluation benchmark in cloud computing based research. First, the workload generation should be non-intrusive, i.e. it should run separate from the approach being tested. Second, the tool should preferably also log a variety of performance and system statistics. These statistics are extremely important to observe the positive or negative effects of the approach under test. Third, the testbed should allow setup within a Virtual Machine (VM), container, cloud instance, or physical host. Fourth, the testbed required could involve a cluster of machines, and the tool should be able to simulate workloads on multiple computers. Finally, the tool should ideally be open source.

This paper introduces an open-source tool called Cloudy that models and runs workloads within cloud instances, VMs, containers, or physical hosts. It is easy to use, and can be downloaded, installed and run without the need for additional configurations in the system and without affecting any other components on the system. The tool uses data traces from more than 16000 machines from Google and Alibaba clusters to provide real-world patterns of memory and CPU usage in real time over multiple days. This ensures a large number of unique workloads that can be run on different machines in a cluster. Additionally, Cloudy features an online visualization dashboard that can be used to observe the CPU and memory usage of a machine, as well as obtain other important performance statistics such as operations per second, number of page faults, etc. over time. Finally, the workload generated can be scaled in terms of both usage and time, giving a finer level of control to the user. It is envisioned that this tool could benefit experimental evaluations of cloud-based research, and provide an easy-to-use standard to compare different approaches. Earlier versions of this tool have been previously used by the authors in [4], [6].

Fig. 1 shows one possible use-case for Cloudy. In order to evaluate a cloud-based framework or approach, a baseline set of performance statistics are obtained by running Cloudy within Infrastructure-as-a-Service (IaaS) instances. Since Cloudy offers more than 16000 unique trace patterns, each instance in this set can run a different real-world workload pattern. Next, Cloudy is restarted with the same workloads as before, this time, along with the approach to be evaluated. The performance statistics are collected again. A comparison of these statistics against the baseline statistics can help researchers evaluate the efficacy of the approach being tested.

The rest of the paper is divided as follows. Section II discusses some existing cloud benchmarking tools and highlights the difference between those tools and Cloudy. Section III describes the implementation, and internals of Cloudy in detail. Section IV reports various experimental results to

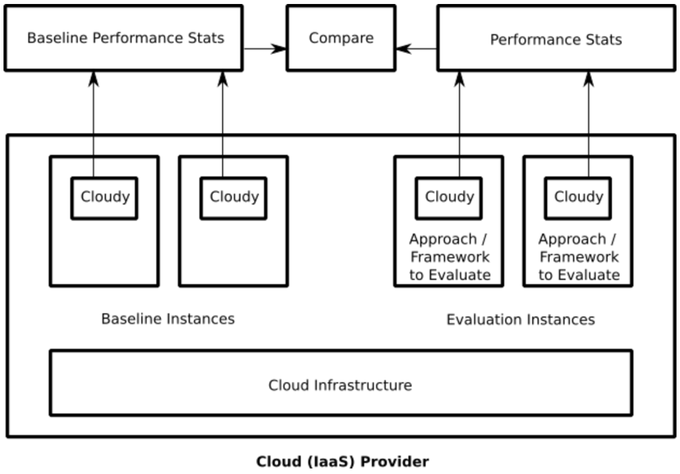
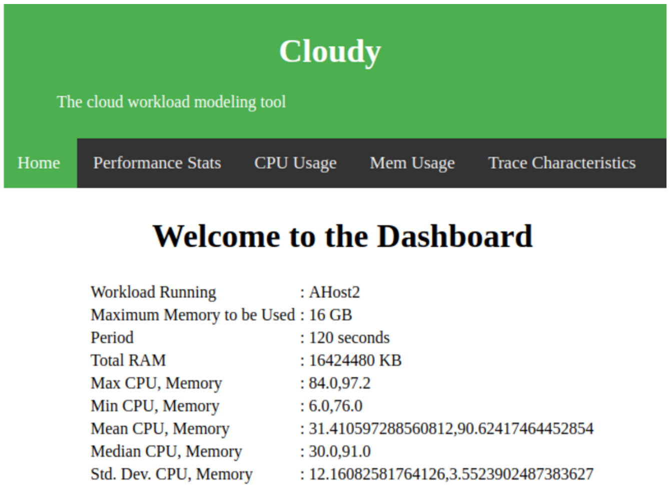


Fig. 1. Use Case of Cloudy.



Welcome to the Dashboard

Workload Running	: AHost2
Maximum Memory to be Used	: 16 GB
Period	: 120 seconds
Total RAM	: 16424480 KB
Max CPU, Memory	: 84.0,97.2
Min CPU, Memory	: 6.0,76.0
Mean CPU, Memory	: 31.410597288560812,90.62417464452854
Median CPU, Memory	: 30.0,91.0
Std. Dev. CPU, Memory	: 12.16082581764126,3.5523902487383627

Fig. 2. Screenshots of Cloudy Web-based Visualization Dashboard.

demonstrate the accuracy and efficacy of Cloudy in recreating the real-world patterns. It also illustrates its utility in further analyzing the traces. Section V notes additional discussions and considerations when using Cloudy. Section VI concludes the paper.

II. Related Work

A number of research endeavors and software exist in literature for evaluating and benchmarking in cloud-based environments. These tools generally fall into one of three categories - benchmarking the cloud itself, testing performance of a required application in different clouds, and general benchmarking tools that solve resource-intensive problems to use resources.

Cloud Bench [7] automates cloud-scale evaluation and benchmarking through the running of controlled experiments, where complex applications are automatically deployed according to user-defined experiment plans. It helps assess the stability, scalability and reliability of different cloud configurations. Similarly, Expertus [8], [9] is a code generation-based approach with the main goal of automating distributed application configuration and testing in IaaS clouds. Cloud Crawler [10], [11] approaches the same problem by providing users with an environment where they can describe a variety of performance evaluation scenarios for a given application. The tool then automatically configures, executes and collects the results of the scenarios described. Cloud WorkBench [12] is another cloud benchmarking service that supports the automatic execution of systematic performance tests in the cloud by leveraging the notion of Infrastructure-as-Code (IaC).

These approaches have a different goal compared to Cloudy. They do not non-obtrusively run a real-world workload in the background. Instead, the workload that will run is the application that a developer intends to move to the cloud. The approaches test and evaluate the given application under different cloud scenarios and offer advice on suitable placement strategies. They are useful for selecting an appropriate configuration of cloud resources for a given application.

RUBiS [13] is a free, open-source auction site prototype modeled after eBay.com. It can be used to evaluate application design patterns and application servers' performance scalability. The website can simulate a real-world load by performing actions such as selling, browsing and bidding. While RUBiS does simulate a real-world application, it is restricted to a scenario consisting of a webserver, specifically for an auction-like site.

Another actively maintained open-source tool that comes with a collection of pre-configured benchmarks is Google's PerfKit Benchmarker [14]. It also offers an optional dashboard for performance analysis. The main goal is to define a canonical set of benchmarks to measure and compare cloud offerings. However, PerfKit does not offer any features that allow generating loads according to real-world patterns.

As opposed to all these approaches, Cloudy focuses on generating CPU and memory load patterns that mirror real-world loads.

III. Method

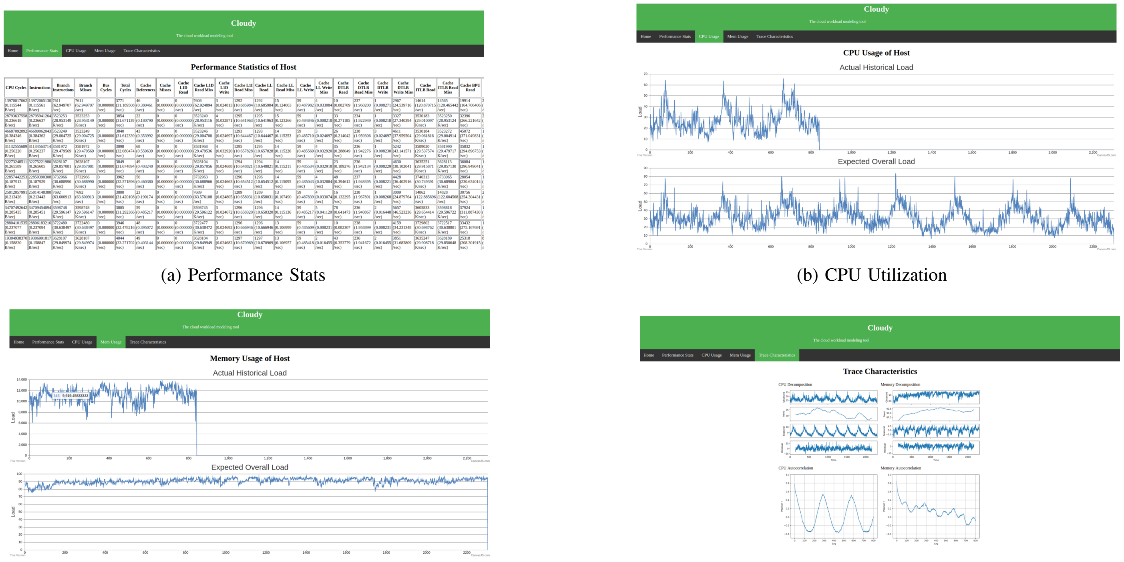
The methodology of Cloudy is discussed in the following subsections from two perspectives: the end-user's perspective (installation, execution, interaction) and implementation (internal components).

A. User's Perspective

From the end-user's perspective, setting up and interacting with Cloudy is a straightforward process. The entire framework with all the required dependencies can be cloned from the Gitlab repository [15] into the VM, local host, or cloud instance of choice. Cloudy can then be installed by running the provided install script (install.sh). Once all the dependencies and file placements are automatically handled, the workload can be started by running the workload.sh script, passing the name of the trace to use (TN), maximum memory to use in GB (MMG) and time scaling in seconds (TSS) as arguments.

./workload.sh *TN* [-m*MMG*] [-t*TSS*]

The argument trace name is the name of the underlying real-world trace that Cloudy will use to generate CPU and



Performance Statistics of Host

CPU Cycles	Instructions	Branch Instructions	Branch Misses	Bus Cycles	Total Cycles	Cache References	Cache Misses	Cache L1D Read	Cache L1D Read Miss	Cache L1D Write	Cache L1I Read Miss	Cache LL Read	Cache LL Read Miss	Cache LL Write	Cache LL Write Miss	Cache DTLB Read	Cache DTLB Read Miss	Cache DTLB Write	Cache DTLB Write Miss	Cache ITLB Read	Cache ITLB Read Miss	Cache BPU Read
13970017062 (0.115544 B/sec)	13972065130 (0.115561 B/sec)	7611 (62.949707 /sec)	7611 (62.949707 /sec)	0 (0.000000 /sec)	3771 (31.189508 /sec)	46 (0.380461 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	7608 (62.924894 /sec)	3 (0.024813 /sec)	1292 (10.685984 /sec)	1292 (10.685984 /sec)	15 (0.124063 /sec)	59 (0.487982 /sec)	4 (0.033084 /sec)	10 (0.082709 /sec)	237 (1.960200 /sec)	1 (0.008271 /sec)	2967 (24.539716 /sec)	14614 (120.870715 /sec)	14565 (120.465442 /sec)	19914 (164.706406 /sec)
28793637558 (0.236618 B/sec)	28795941264 (0.236637 B/sec)	3523253 (28.953149 K/sec)	3523253 (28.953149 K/sec)	0 (0.000000 /sec)	3854 (31.671139 /sec)	22 (0.180790 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3523249 (28.953116 K/sec)	4 (0.032871 /sec)	1295 (10.641963 /sec)	1295 (10.641963 /sec)	15 (0.123266 /sec)	59 (0.484846 /sec)	1 (0.008218 /sec)	33 (0.271185 /sec)	234 (1.922949 /sec)	1 (0.008218 /sec)	3327 (27.340394 /sec)	3530183 (29.010097 K/sec)	3523250 (28.953124 K/sec)	32396 (266.221642 /sec)
46687092892 (0.384346 B/sec)	46689062043 (0.384362 B/sec)	3523249 (29.004725 K/sec)	3523249 (29.004725 K/sec)	0 (0.000000 /sec)	3840 (31.612339 /sec)	43 (0.353992 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3523246 (29.004700 K/sec)	3 (0.024697 /sec)	1293 (10.644467 /sec)	1293 (10.644467 /sec)	14 (0.115253 /sec)	59 (0.485710 /sec)	3 (0.024697 /sec)	26 (0.214042 /sec)	238 (1.959306 /sec)	3 (0.024697 /sec)	4611 (37.959504 /sec)	3530184 (29.061816 K/sec)	3523272 (29.004914 K/sec)	45072 (371.049831 /sec)
31132555689 (0.256220 B/sec)	31134563714 (0.256237 B/sec)	3581972 (29.479569 K/sec)	3581972 (29.479569 K/sec)	0 (0.000000 /sec)	3898 (32.080474 /sec)	68 (0.559639 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3581968 (29.479536 K/sec)	4 (0.032920 /sec)	1295 (10.657828 /sec)	1295 (10.657828 /sec)	14 (0.115220 /sec)	59 (0.485569 /sec)	4 (0.032920 /sec)	35 (0.288049 /sec)	236 (1.942276 /sec)	1 (0.008230 /sec)	5242 (43.141571 /sec)	3589020 (29.537574 K/sec)	3581990 (29.479717 K/sec)	35832 (294.896753 /sec)
32273248511 (0.265589 B/sec)	32275221980 (0.265605 B/sec)	3628107 (29.857081 K/sec)	3628107 (29.857081 K/sec)	0 (0.000000 /sec)	3849 (31.674894 /sec)	49 (0.403240 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3628104 (29.857056 K/sec)	3 (0.024688 /sec)	1294 (10.648821 /sec)	1294 (10.648821 /sec)	14 (0.115211 /sec)	59 (0.485534 /sec)	4 (0.032918 /sec)	23 (0.189276 /sec)	236 (1.942134 /sec)	1 (0.008229 /sec)	4630 (38.102041 /sec)	3635251 (29.915871 K/sec)	3628113 (29.857130 K/sec)	36084 (296.949041 /sec)
22857442253 (0.187913 B/sec)	22859390308 (0.187929 B/sec)	3732966 (30.688990 K/sec)	3732966 (30.688990 K/sec)	0 (0.000000 /sec)	3962 (32.571896 /sec)	56 (0.460380 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3732963 (30.688966 K/sec)	3 (0.024663 /sec)	1296 (10.654512 /sec)	1296 (10.654512 /sec)	14 (0.115095 /sec)	59 (0.485043 /sec)	4 (0.032884 /sec)	48 (0.394612 /sec)	237 (1.948395 /sec)	1 (0.008221 /sec)	4428 (36.402916 /sec)	3740313 (30.749391 K/sec)	3733065 (30.689804 K/sec)	28054 (230.634014 /sec)
25812057991 (0.213426 B/sec)	25814148380 (0.213443 B/sec)	7692 (63.600913 /sec)	7692 (63.600913 /sec)	0 (0.000000 /sec)	3800 (31.420108 /sec)	23 (0.190174 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	7689 (63.576108 /sec)	3 (0.024805 /sec)	1289 (10.658031 /sec)	1289 (10.658031 /sec)	13 (0.107490 /sec)	59 (0.487839 /sec)	4 (0.033074 /sec)	16 (0.132295 /sec)	238 (1.967891 /sec)	1 (0.008268 /sec)	3009 (24.879764 /sec)	14862 (122.885696 /sec)	14828 (122.604568 /sec)	30756 (254.304431 /sec)
34707492642 (0.285435 B/sec)	34709454094 (0.285451 B/sec)	3598748 (29.596147 K/sec)	3598748 (29.596147 K/sec)	0 (0.000000 /sec)	3805 (31.292366 /sec)	59 (0.485217 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3598745 (29.596122 K/sec)	3 (0.024672 /sec)	1296 (10.658320 /sec)	1296 (10.658320 /sec)	14 (0.115136 /sec)	59 (0.485217 /sec)	5 (0.041120 /sec)	78 (0.641473 /sec)	236 (1.940867 /sec)	2 (0.016448 /sec)	5657 (46.523236 /sec)	3605833 (29.654414 K/sec)	3598818 (29.596722 K/sec)	37924 (311.887430 /sec)
28804101535 (0.237077 B/sec)	28806183216 (0.237094 B/sec)	3722480 (30.638497 K/sec)	3722480 (30.638497 K/sec)	0 (0.000000 /sec)	3946 (32.478216 /sec)	48 (0.395072 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3722477 (30.638472 K/sec)	3 (0.024692 /sec)	1296 (10.666946 /sec)	1296 (10.666946 /sec)	13 (0.106999 /sec)	59 (0.485609 /sec)	1 (0.008231 /sec)	10 (0.082307 /sec)	238 (1.958899 /sec)	1 (0.008231 /sec)	4159 (34.231348 /sec)	3729802 (30.698762 K/sec)	3722517 (30.638801 K/sec)	33432 (275.167691 /sec)
19304938370 (0.158830 B/sec)	19306991817 (0.158847 B/sec)	3628107 (29.849974 K/sec)	3628107 (29.849974 K/sec)	0 (0.000000 /sec)	4044 (33.271702 /sec)	49 (0.403144 /sec)	0 (0.000000 /sec)	0 (0.000000 /sec)	3628104 (29.849949 K/sec)	3 (0.024682 /sec)	1297 (10.670969 /sec)	1297 (10.670969 /sec)	13 (0.106957 /sec)	59 (0.485418 /sec)	2 (0.016455 /sec)	43 (0.353779 /sec)	236 (1.941672 /sec)	2 (0.016455 /sec)	3851 (31.683809 /sec)	3635247 (29.908718 K/sec)	3628189 (29.850648 K/sec)	25318 (208.301915 /sec)

(a) Performance Stats

(b) CPU Utilization

(c) Memory Utilization

(d) Trace Characteristics

Fig. 3. Screenshots of Cloudy Web-based Visualization Component.

memory usage. It could be the name any one of the 16000 traces available in the repository. This argument is mandatory. The maximum memory to use is an important parameter that can be tweaked based on testing requirements. By default, when generating a pattern, Cloudy will use up to all the available memory. However, specifying a maximum will restrict Cloudy from zero to the maximum memory specified. It is important to note, that in either case, the pattern generated will look exactly the same and follow the underlying trace chosen, it will simply be scaled to the maximum memory specified. Similarly, the time scaling pattern allows the user to specify the duration of the entire workload. By default, each data point in the underlying trace is considered to be at 5 minutes (which is the actual time frame). However, specifying a different time scale (for example 120 seconds) will make Cloudy consider each underlying data point at the new time scale (every 120 seconds in the example). Again, the overall pattern of the trace will not be affected, instead this will simply stretch or shrink the entire trace in time.

Once suitable arguments are chosen (or left to default), Cloudy starts utilizing CPU and memory over time, according to the trace chosen.

The installer also sets up a webserver and front-end dashboard on the same machine, which can be accessed by browsing to the ip address of the machine, as long as port 80 is accessible. Fig. 2 is a screenshot of Cloudy's dashboard. This is the landing page of the ip address of a machine running Cloudy. The dashboard reports summarized statistics on the current state of the machine, and the underlying trace being used. This information includes the amount of memory available, as well as the name of the trace being run, and the maximum, minimum, median, mean, and standard deviation of both, memory and CPU usage of the trace.

The web-based interface also provides other detailed analysis of the system and trace being used. Fig. 3 shows screenshots of the remaining four sections of the visualization. Fig. 3a shows a report of the performance metrics collected through the entire run of the workload. These include the CPU cycles, page fault count, context switches, cache-related statistics, etc. and are also recorded in a logfile. Fig. 3b, 3c show the real time graphs for CPU and memory usage respectively. These sections also report the graphs for the entire trace for both CPU and memory usage. Finally, Fig. 3d calculates and generates statistics to evaluate the overall trace. These statistics include decomposition of the trace into trend, seasonal, and residual components, as well as autocorrelation plots. The statistics are generated for both the CPU and memory traces. These features are discussed in more detail in the section on Results.

Currently, the install script provided supports Ubuntu-based AWS ec2 instances. However, Cloudy can still be run without any modification on most Linux-based machines within different commercial cloud providers (such as Google's Compute Engine).

B. Implementation

Fig. 4 shows an overview of Cloudy. There are three main components of Cloudy, which include the workload trace, the load generator, and the web-based visualization component. As depicted in the figure, the workload traces are individual files, with the percent memory and CPU usage of 16000 machines, stored on a remote file hosting server. One of these traces is selected for each run of the model. When Cloudy is run, the initialization step downloads the trace file specified by the trace name (TN) argument onto the VM or container being tested. This trace file is picked up by the load generator, which follows the trace to generate matching memory and CPU loads over time. Finally, the visualization component, which also exists in the VM or container, can be accessed by any browser over the internet through the ip address of the VM or container to view details and statistics about the workload. The next subsections discuss the details of each of these components.

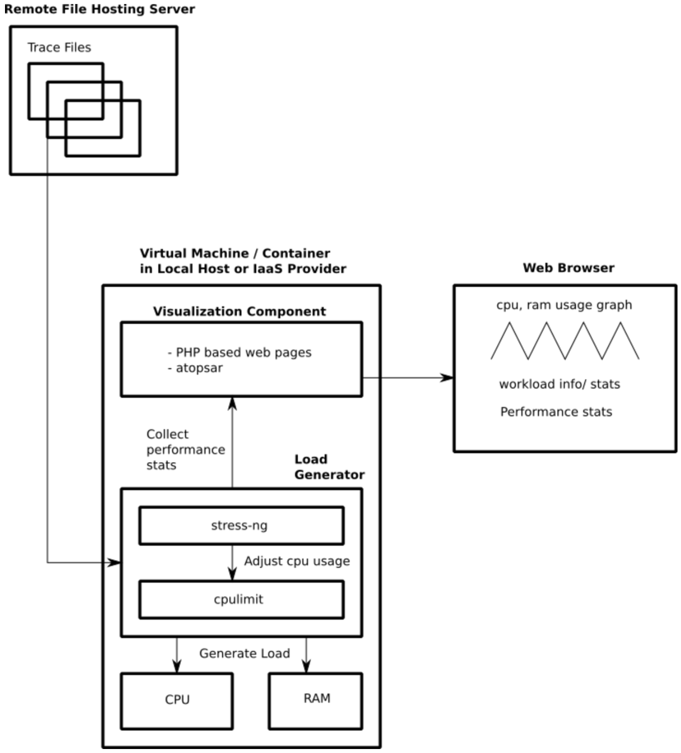


Fig. 4. Overview of Cloudy.

1) Data Traces: The data traces, which are stored on a remote server, hold the CPU and memory usage over time of one machine each. The files are structured so that each line has comma separated CPU and memory usage (in percentage of total) within a 5 minute period. There are a total of 16000 traces, 12000 of which belong to the Google cluster [16], and 4000 belong to the AliBaba cluster [17]. Cloudy uses these trace files as a guide to generating workloads. Next, the two cluster traces, and the mechanism used for extracting the relevant traces from the two datasets are described.

Google Data Traces: The Google cluster data trace consists of 29 days' worth of logs for about 12000 machines, from a Google cluster in a datacenter in the US, starting at 19:00 EDT on Sunday, May 1, 2011. In this context, a Google cluster is a set of machines, packed into racks, and connected by a high-bandwidth cluster network. A set of these machines (cell) is allocated work by a cluster-management system. Work arrives at a cell in the form of jobs which are comprised of one or more tasks, and these tasks run on machines. Each task is a Linux program made up of multiple processes and runs on a single machine. The usage data for the tasks were collected from the management system and the individual machines. The data is represented as percent CPU and memory usage of each task at 5 minute intervals.

The trace contains a number of tables describing different information. These tables include:

- Machine events - describes addition, removal, updates of machines
- Machine attributes - describes machine properties such as kernel version, clock speed, etc.
- Job events - describes when each job was submitted, scheduled, run, etc.
- Task events - describes which machines tasks are located in, resources requested, etc.
- Task constraints - describes constraints on placement of tasks, if any
- Task resource usage - describes mean CPU usage, memory usage, disk I/O time, etc. for each task at each time instance

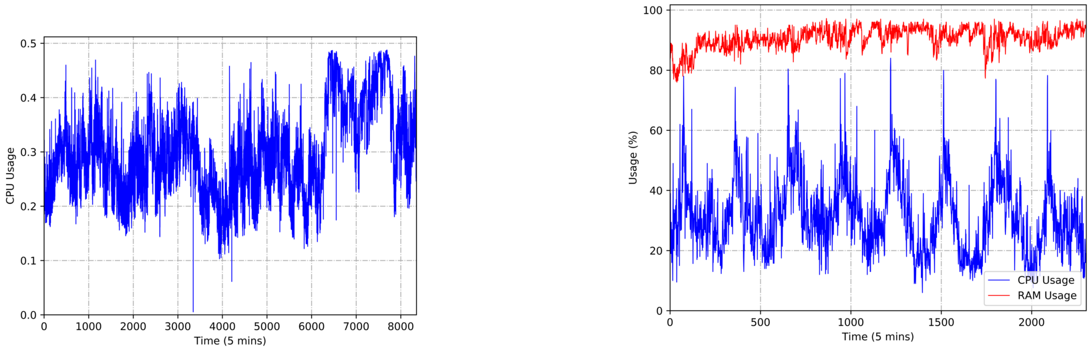
Of these tables, the task resource usage is of particular interest. Since the Google cluster data does not directly provide the CPU and memory usage on a particular machine, it has to be calculated. For a given instance in time, this is done by adding up the usages of all the tasks residing on the machine at that time. A python script was written to collate all the tasks on the same machine, and then calculate the sum of their usages at each time interval (5 minutes). The final traces are stored in separate files for each machine. The files are named GHost0 to GHost11999. The end result is a set of 12000 files with 8352 data points each (29 days at 5 minute intervals) specifying the percent of CPU and memory usage. Fig. 5a shows the CPU usage of a sample host from the Google cluster dataset for the 29 days of the trace.

Alibaba Cluster Data Traces: The Alibaba cluster data trace includes about 4000 machines for the Alibaba website, during a period of 8 days, and consists of six tables (each is a file). These tables include:

- machine_meta.csv: the meta info and event information of machines
- machine_usage.csv: the resource usage of each machine
- container_meta.csv: the meta info and event information of containers
- container_usage.csv: the resource usage of each container
- batch_instance.csv: information about instances in the batch workloads
- batch_task.csv: information about instances in the batch workloads

As opposed to the Google cluster data, the Alibaba data traces directly specify the percent CPU and memory usage of each machine at a given time. This can be obtained from the machine_usage.csv file. Using a python script, the usages of the machines were separated, and arranged in 5 minute intervals. The final traces are stored as separate files for each machine. The files are named AHost0 to AHost3999. The end result is a set of 4000 files with 2304 data points each (8 days at 5 minute intervals) specifying the percent of CPU and memory usage. Fig. 5b shows both the CPU and memory usage of a sample host from the Alibaba traces for the 8 days of the trace.

2) Load Generation: The load generator runs in a loop, reading a pair of CPU and memory values from the given trace file periodically. The period is dictated by the timescaling



(a) Google Host

(b) Alibaba Host

Fig. 5. Sample Hosts CPU and Memory Usage.

(TSS) argument given when running Cloudy. As mentioned, by default, the load generator reads a new pair of values every 5 minutes. Each period, the generator aims to generate a physical workload that matches both the CPU and memory usage specified by the pair of values. In order to generate this workload, a utility must be chosen that solves a generic problem, thus utilizing CPU and memory. For example, allocating and modifying large arrays can be used to simulate memory usage, while linear algebra solvers can simulate CPU usage. While it is fairly trivial to run a utility that simulates a certain amount of memory usage or CPU usage, it is extremely difficult to choose a single tool that utilizes an exact, arbitrary amount of both memory and CPU usages as required.

Cloudy approaches load generation in two steps. At any point in time, first the memory load required for the current period is generated by running a suitable utility. However, any such utility, will end up working at full available CPU capacity. Therefore, in the next step, a limit on the amount of CPU that can be used is applied to the running utility to match the CPU usage required for the current period.

To achieve the first step of memory load generation, the benchmarking utility stress-ng is used. This utility allows stress-testing a system in a number of selectable ways. Stress-ng has a variety of stressors including floating point, integer, or bit manipulation for CPU, i/o devices, network, schedulers, etc. Cloudy utilizes stress-ng's memory stressor to generate controlled, memory intensive loads. The memory stressor can be given a size of memory to use, and the stressor continuously calls mmap for the specified size and writes to the allocated memory. Since the trace files provide memory usage as a percentage, the load generator calculates the size of the memory to use based on the maximum memory (MMG) argument (if given) or the total memory available (default).

Once the required amount of memory is being used, the second step begins. The program cpulimit can be given a CPU usage percentage, and the PID of a process to limit the real CPU usage of the process to the desired percentage. Using this, the load generator limits the CPU usage of the running stress-ng process to the usage required for the current period.

At this point, both the CPU and memory usage of the machine match the values specified by the trace for the current period. These usages continue until the next period, when the current stress-ng process terminates, and the previous two steps are repeated for the next pair of values from the trace file.

3) Visualization: The Visualization component of Cloudy consists of some backend scripts for data collection and calculation and a frontend. The statistics that are recorded for visualization, are all returned from the stress-ng utility, and are collected at the end of each period. These include the operations per second, page fault count, etc. and are recorded in a logfile while stress-ng runs.

Additionally, to view the actual CPU and memory usage of the VM or container in real time, the program atopsar is used. Atopsar can report statistics on a system level and return periodic information about the usage.

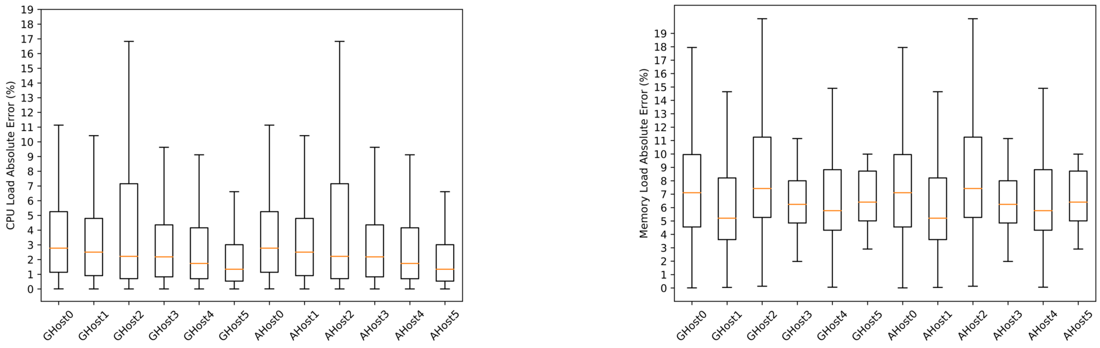
In order to retrieve the information in a suitable fashion, the logging features of both stress-ng and atopsar have been modified. The modifications only include changes to the output formats so that the outputs can be redirected to the logfiles, without the need for additional scripts to clean the data.

Finally, a backend Python script is used to calculate and plot the decompositions and autocorrelation values for both CPU and memory from the current trace file.

The front end of the visualization component is built using PHP. When installing Cloudy from the git repository, the entire Visualization component is included, and the front-end as well as the modified versions of stress-ng and atopsar are automatically installed.

IV. Experimental Evaluations and Results

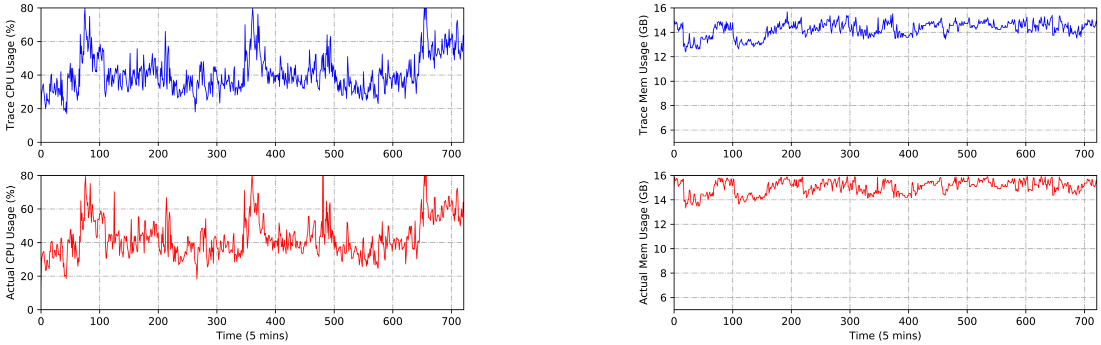
In order to evaluate Cloudy, multiple runs with different traces were performed on Amazon Web Services' ec2 instances (t2.xlarge: Ubuntu 18.04, 4 cores, 16 GB RAM, 40 GB EBS). For the experiments in this paper, the maximum memory to use was set to 16 GB, and the scaling was at the default of 5 minutes. Currently, 2000 traces are available in the gitlab repository. These include 1000 traces each from Google and Alibaba workloads (GHost0 to GHost999 and AHost0 to AHost999). The experiments that follow use samples from these 2000 traces. All the 16000 traces are currently being placed on a suitable ftp server, and are available on request.



(a) CPU

(b) Memory

Fig. 6. Absolute Actual vs. Trace Load Error.



(a) CPU

(b) Memory

Fig. 7. Actual vs. Trace Usage.

The following subsections evaluate two aspects: the accuracy of Cloudy when recreating patterns from underlying traces, and characteristics of the traces that can be gleaned using Cloudy.

A. Cloudy Evaluation

One of the important aspects of evaluating the efficacy of Cloudy is to analyze how closely the generated CPU and memory usages follow the usages in the underlying data traces. For these experiments, 12 traces (AHost0-5 and GHost0-5) were separately run for their entire duration, and evaluated on the ec2 instances. This implies that Cloudy was run for 29 days for each of the GHost traces, and 8 days for each of the AHost traces. The logged actual CPU and memory usage over these 12 runs was then compared to the usages according to the underlying traces. The absolute error at each period for each host was calculated as $abs(usage_{actual} - usage_{trace})$ Fig. 6 plots boxplots of the absolute errors for each of the 12 hosts.

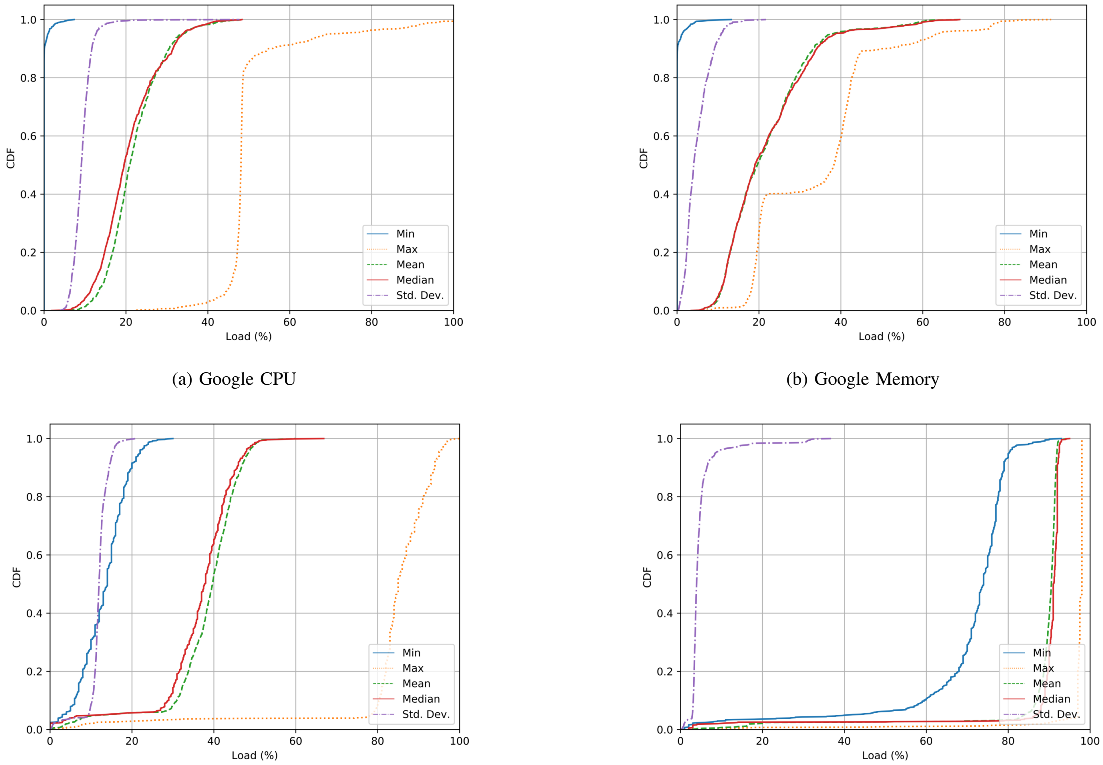
The plots show that for the 12 hosts, the median CPU error is mostly at about 2-3%. At worst, the generated CPU usage deviates by about 17% for AHost2. The few extremely high error moments can be attributed to external factors, such as the underlying OS performing system tasks, etc. Even then, for AHost2, 75% of the errors are at or below 7% and 50% of the errors are at or below about 2%. Similarly, the median memory error stays in the range of 5-7% for all 12 hosts. This demonstrates that generally, with an error of less than 7%, Cloudy accurately recreates the CPU and memory usage of the underlying trace.

The average CPU and memory errors for 12 hosts are given in Table I.

TABLE I. Actual vs. Trace Load Absolute Errors

Trace Name	CPU (%)	Memory (%)
AHost0	7.96	6.07
AHost1	6.83	17.53
AHost2	8.48	5.51
AHost3	9.23	4.79
AHost4	4.89	4.97
AHost5	7.72	4.91
GHost0	3.95	8.07
GHost1	3.88	5.97
GHost2	5.71	9.05
GHost3	3.34	6.31
GHost4	3.44	6.52
GHost5	2.19	6.43

The figure and table indicate that the percent memory usage generally has a median error of about 6%. To put this in absolute memory terms, since 16 GB instances were used, 6% equates to about 0.96 GB. This additional memory usage corresponds to the memory requirements of the underlying



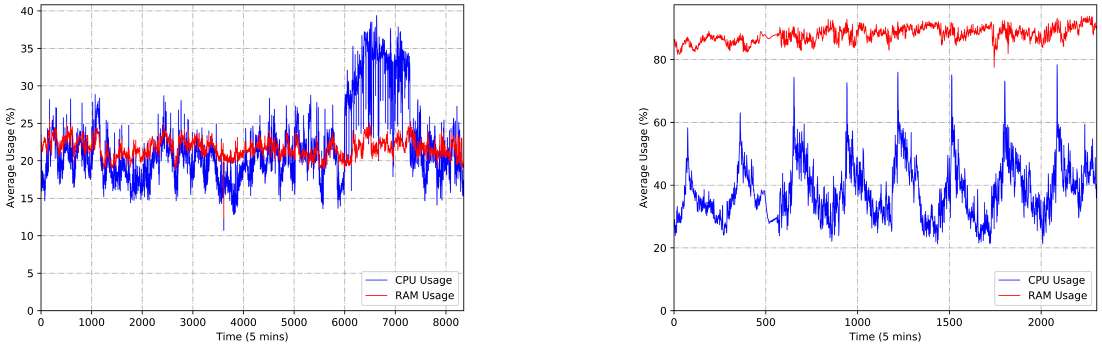
(a) Google CPU

(b) Google Memory

(c) AliBaba CPU

(d) AliBaba Memory

Fig. 8. CDFs of Min, Max, Mean, Median, and Std. Dev. for CPU and Memory of all Workloads.



(a) Google

(b) AliBaba

Fig. 9. Average CPU and Memory usage of 1000 machines over time.

operating system (OS) and its processes. If greater accuracy in memory usage is required, the maximum memory to use argument can be tweaked while starting Cloudy, to accommodate for the memory requirements for the underlying OS. For reference, Fig. 7 shows the actual load generated vs trace load for a sample workload (AHost4). It can be observed that the generated load closely matches the pattern of the load indicated by the underlying trace.

B. *Workload Characteristics*

This subsection evaluates and discusses the behavior of the underlying traces that Cloudy follows to generate the workloads. There are two main purposes of these evaluations. First, to provide the reader with an idea of the nature and type of the underlying traces. Second, to demonstrate the various types of analysis that can be performed on the workloads when using Cloudy.

In order to meet these goals, the following subsections discuss some aggregated statistics such as minimum, maximum, mean loads, and standard deviations, as well as seasonal decomposition of the loads, autocorrelation of the loads, and cross-correlation of CPU loads with memory loads. All of these characteristics for a running workload can be viewed through the visualization component of Cloudy. For this set of experiments, all 2000 traces were used. The CPU and memory usages were separated, resulting in 4000 total traces.

1) Aggregated Statistics: Fig. 8 shows four CDFs that summarize the aggregated values of the Google and Alibaba traces. The reported parameters are the maximum, minimum, mean, median, and standard deviation values over the entire duration of the traces. For the Google CPU traces, the average maximum and minimum values are 55% and 0.16%, respectively, while for Google memory traces, the average maximum and minimum values are 34.45% and 0.21%, respectively. For the Alibaba CPU traces, the average maximum and minimum values are 83.7% and 13.13%, respectively, while for the Alibaba memory traces, the average maximum and minimum values are 96.54% and 69.81%, respectively. From the figures, the standard deviations indicate, that in both Google and Alibaba traces, memory usage is generally less variable around its mean, as opposed to CPU usage that varies substantially within a single trace. Further, the Alibaba traces in general, show higher memory and CPU usages as opposed to the Google traces. Finally, the Alibaba traces show substantially high memory usage for most traces.

To offer an overall view of the traces, Fig. 9 shows the average CPU and memory usage for both Alibaba and Google traces at each instance of time. It can be seen that over all the observed workloads, the Alibaba CPU traces have a more obvious pattern than the Google traces. The memory traces in both cases, does not show an apparent pattern. However, as suggested before, it can be deduced that the Alibaba memory traces utilize more memory than the Google memory traces.

2) Seasonality and Trends: In order to analyze the periodic nature of the traces, as well as any inherent patterns, all the Google and Alibaba traces were decomposed into their trend, seasonal, and residual components. Fig. 10 shows one sample each of the Google CPU, Google memory, Alibaba CPU, and Alibaba memory traces. Decompositions for all the traces can be viewed through Cloudy. It is important to note that the x-axis scales for Google and Alibaba are different since their durations are different (24 days and 8 days respectively). Based on auditing the decompositions, similar trends and patterns exist across all Google and Alibaba traces. The Alibaba CPU traces demonstrate a clear seasonal pattern corresponding to one day. While the other three types of traces also demonstrate a seasonal pattern, the residual components for them do not seem to be simply noise (especially for the memory traces). This suggests the need for some further investigation into the inherent patterns within the memory traces.

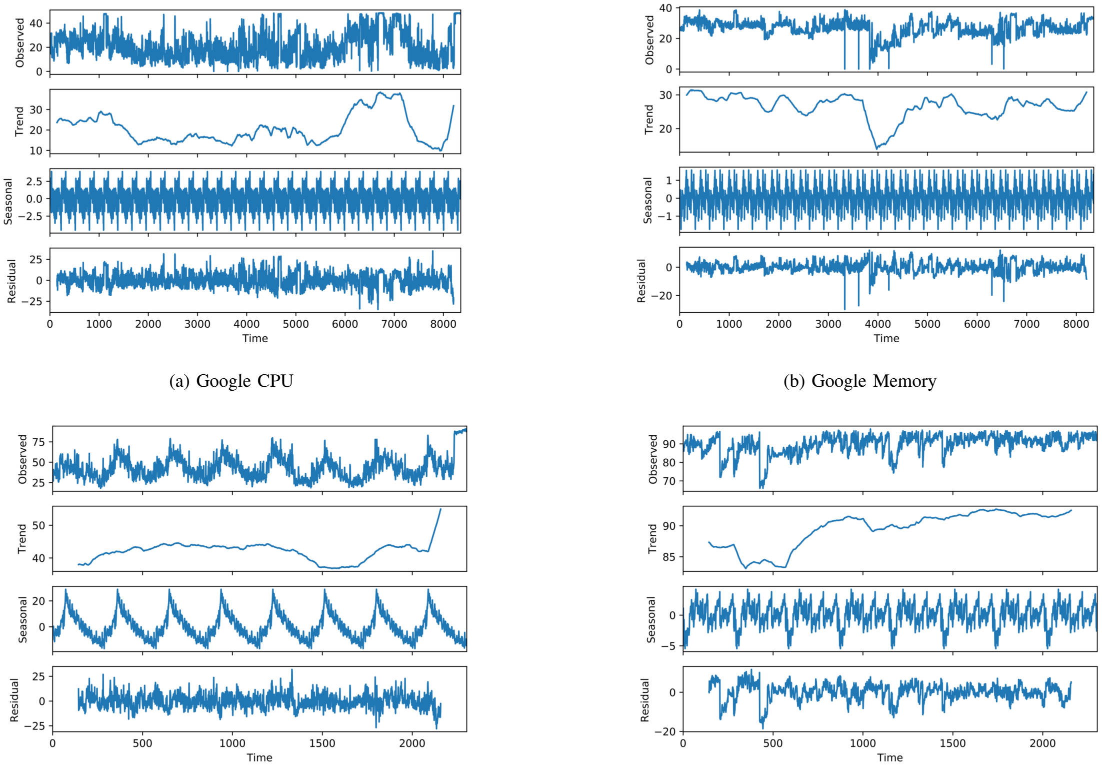
3) Autocorelation and Cross-corelation: In order to further understand whether any patterns exist in the traces, all 2000 traces were analyzed for autocorrelation. After calculating the autocorrelation function (ACF) values for each trace up to lag 800, the maximum value not at lag 0 were logged. Fig. 11a shows a boxplot of these maximum ACF values for all the traces, separated by type. The figure can provide a general idea of the amount of autocorrelation that exists on an average in these traces. It can be observed that the traces from the Alibaba CPU have higher median maximum ACF values as opposed to the other types of traces. This indicates higher autocorrelation in the Alibaba CPU traces. Similarly, on an average, lower autocorrelations can be seen in the Alibaba memory traces. The median maximum ACFs for Google CPU, Google memory, Alibaba CPU, and Alibaba memory traces are about 0.35, 0.35, 0.5, and 0.25 respectively. The observation supports the analysis from the previous section, that demonstrated high seasonality in the Alibaba CPU traces. This can be used as a starting point for further analysis into the patterns and predictabilty of the traces. Fig. 12 shows the autocorrelation plots for sample traces (one Google and one Alibaba). These plots are available from the Visualization component of Cloudy. The Alibaba CPU trace shows obvious, high peaks at non-zero lags, indicating a high degree of autocorrelation. While the Alibaba memory plot in this sample also shows a high degree of autocorrelation, that is not generally true for most other Alibaba memory traces. The Google traces do not show any prominent autocorrelation at any lag.

Another important aspect to consider for a workload on a machine is the relationship between the CPU and memory usage. Intuitively, since a running program is working with both CPU and memory, it stands to reason that for a given workload, there could be some positive or negative (in some cases) correlation between usages of the two. This analysis can prove extremely beneficial in a variety of load predicting algorithms, and can potentially provide better results than predicting on CPU or memory alone. With this in mind, cross-correlation between CPU and memory for the 2000 traces for up to lag 800 is reported. Similar to the analysis with autocorrelation, for each of the 2000 traces, the maximum value of the cross-correlation function (CCF) not at lag 0 were logged. Fig. 11b shows a boxplot of these maximum CCF values for all the traces. In this case, it can be seen that overall, there does not seem to be a strong cross-correlation between memory and CPU for either the Google or Alibaba traces. The Google traces have a slightly higher cross-correlation between memory and CPU usage, with a median maximum CCF of about 0.3, and 75% of the traces showing maximum CCF under 0.4. Compared to this, the Alibaba traces have a median maximum CCF of about 0.23, and 75% of the traces showing maximum CCF under 0.25.

V. Discussions and Future Work

The experimental results show an average memory error of approximately 1 GB. This is an important aspect to consider. The reason for this error is the memory that the underlying OS requires for its own purposes, even without Cloudy running. Typically, for the ec2 Ubuntu instances, this corresponds to a little under 1 GB. It is therefore recommended that when running Cloudy, the maximum memory to be used is specified keeping the underlying OS's requirement in mind. For example, in the scenarios described previously, Cloudy should be run at a maximum of 15 GB memory (instead of 16 GB). This will ensure that the resultant memory load matches the trace load even more closely, with negligible errors.

There are three aspects of Cloudy that are currently being worked on to make the tool more universal. The first aspect deals with the statistics logged and displayed. Currently, the performance statistics provided are recorded via stress-ng, and



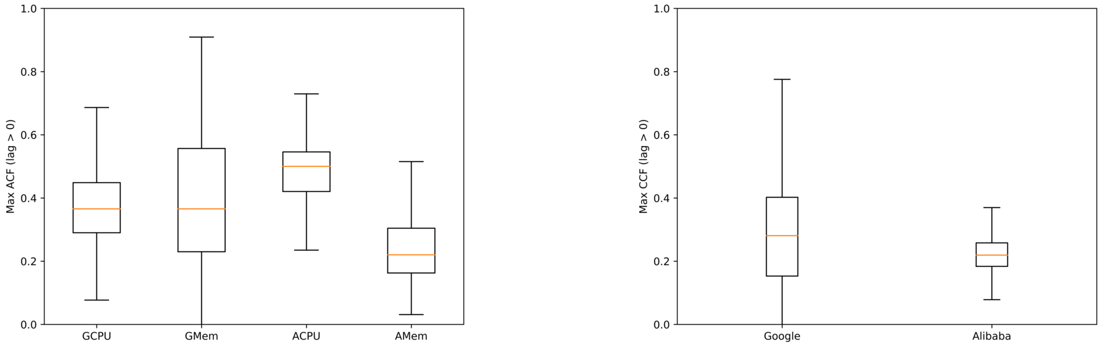
(a) Google CPU

(b) Google Memory

(c) AliBaba CPU

(d) AliBaba Memory

Fig. 10. Sample Decompositions of Workloads.



(a) Autocorrelation

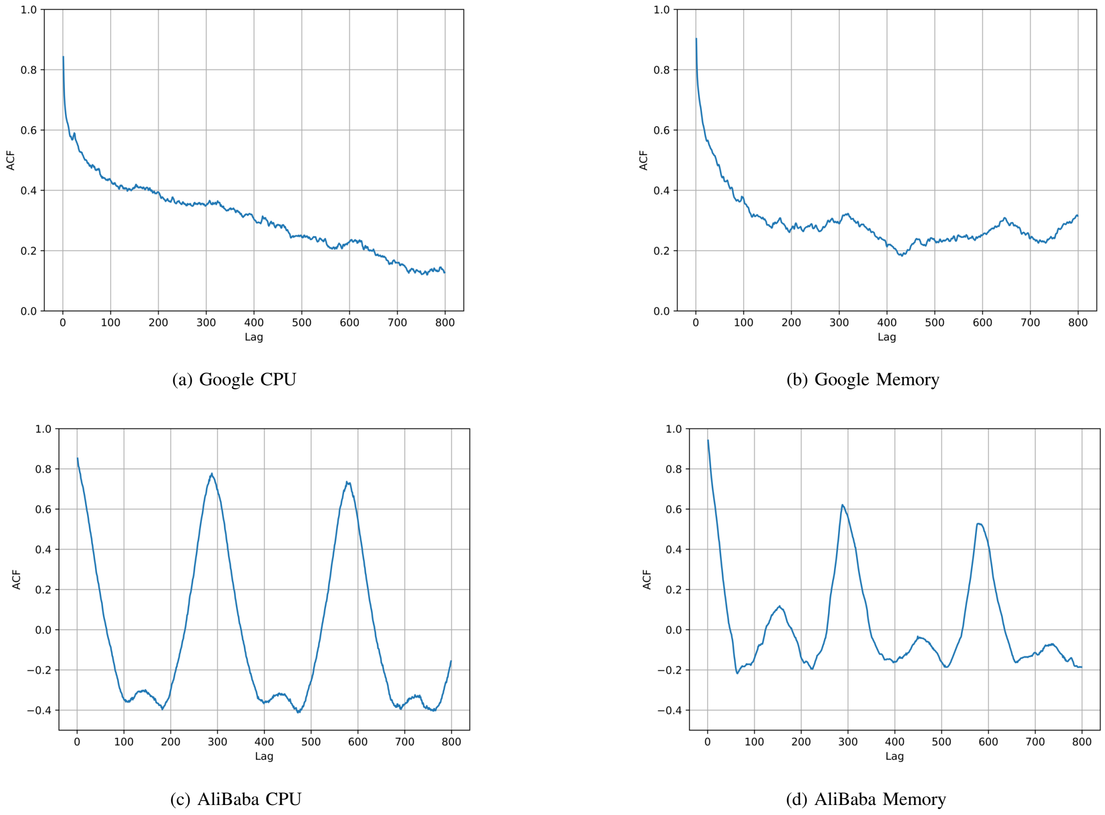
(b) Cross-correlation (CPU and memory usage)

Fig. 11. Maximum Correlation Function Values.

have to be used in that context. However, with only some slight additions and no changes to the behavior of the framework, other desired system-wide parameters can be recorded and displayed. Based on user input after release, the next update of Cloudy shall include other statistics as requested.

The second aspect is the utility used to create the load on memory, viz. stress-ng. Again, without any major changes to the behavior and code of the framework, any utility can be used to generate the memory load. For example, typical programs that are used to generate memory loads include array sorters, linear algebra solvers, matrix operators, etc. The next iteration of Cloudy aims to offer multiple stress-ng-like utilities that users can choose from, when running Cloudy. This will empower the user to select a work that is more representative of the types of load they envision in context of their testbed.

Finally, Cloudy has been tested and validated on Ubuntu based AWS ec2 instances. However, there is no part of the framework that prevents it from being run on any Linux-based distribution. Automatic install scripts for other distributions and cloud providers are currently been implemented, and shall be added to the git repository.



(a) Google CPU

(b) Google Memory

(c) AliBaba CPU

(d) AliBaba Memory

Fig. 12. Sample Autocorrelation of Workloads.

VI. Conclusion

This paper introduced a free, open-source, workload generator called Cloudy. The generator is aimed at researchers in cloud computing who need a testbed to evaluate their own research ideas. Cloudy is easy to install, non-intrusive, and can be used to quickly simulate real-world CPU and memory usage patterns in VMs, containers, cloud instances, or local machines. Through extensive experimental evaluations it was demonstrated that using Cloudy, the CPU and memory usage on a machine can closely follow one of 16000 real-world usage traces. Additional evaluations demonstrated the various analysis features of Cloudy that can allow users to further enhance their understanding of the underlying real-world loads, rather than running a black-box generator.

References

[1] M. R. Hines and K. Gopalan, “Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 51–60.

[2] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman, “Optimized pre-copy live migration for memory intensive applications,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 40.

[3] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Autonomic vertical elasticity of docker containers with elasticdocker,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 472–479.

[4] K. Qazi and S. Romero, “Remote memory swapping for virtual machines in commercial infrastructure-as-a-service,” in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE, 2019, pp. 1–8.

[5] G. Moltó, M. Caballer, and C. De Alfonso, “Automatic memory-based vertical elasticity and oversubscription on cloud platforms,” *Future Generation Computer Systems*, vol. 56, pp. 1–10, 2016.

[6] K. Qazi, “Vertelas - Automated user-controlled vertical elasticity in existing commercial clouds,” in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE, 2019, pp. 1–8.

[7] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. Da Silva, “Cloudbench: Experiment automation for cloud environments,” in *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2013, pp. 302–311.

[8] D. Jayasinghe, J. Kimball, S. Choudhary, T. Zhu, and C. Pu, “An automated approach to create, store, and analyze large-scale experimental data in clouds,” in *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE, 2013, pp. 357–364.

[9] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu, “Expertus: A generator approach to automate performance testing in IaaS clouds,” in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 115–122.

[10] M. Cunha, N. Mendonca, and A. Sampaio, “A declarative environment for automatic performance evaluation in IaaS clouds,” in *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013, pp. 285–292.

[11] M. Cunha, N. Mendonça, and A. Sampaio, “Cloud Crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 1, p. e3825, 2017.

[12] J. Scheuner and P. Leitner, “Performance benchmarking of infrastructure-as-a-service (IaaS) clouds with cloud workbench,” in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. ACM, 2019, pp. 53–56.

[13] "RUBiS," Aug 2019, posted at https://github.com/uillianluiz/RUBiS. (Accessed: May 2020).

[14] Google, "PerfKit benchmarker," github, 2020, posted at https://github.com/GoogleCloudPlatform/PerfKitBenchmarker (Accessed: May 2020).

[15] K. Qazi, "Cloudy," gitlab, 2020, posted at https://gitlab.com/kashifqazi/cloudy.

[16] J. Wilkes, "More Google cluster data," Google research blog, Nov. 2011, posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html (Accessed: May 2020).

[17] Alibaba, "Alibaba production cluster data v2018," github, 2018, posted at https://github.com/alibaba/clusterdata/tree/v2018 (Accessed: May 2020).