CloudPerf: A Performance Test Framework for Distributed and Dynamic Multi-Tenant Environments

Nicolas Michael
nicolas.michael@oracle.com

Nitin Ramannavar
nitin.ramannavar@oracle.com

Yixiao Shen
yixiao.shen@oracle.com

Sheetal Patil
sheetal.patil@oracle.com

Jan-Lung Sung
janlung.sung@oracle.com

Oracle Inc.
4180 Network Circle
Santa Clara, California 95054

ABSTRACT

The evolution of cloud-computing imposes many challenges on performance testing and requires not only a different approach and methodology of performance evaluation and analysis, but also specialized tools and frameworks to support such work. In traditional performance testing, typically a single workload was run against a static test configuration. The main metrics derived from such experiments included throughput, response times, and system utilization at steady-state. While this may have been sufficient in the past, where in many cases a single application was run on dedicated hardware, this approach is no longer suitable for cloud-based deployments. Whether private or public cloud, such environments typically host a variety of applications on distributed shared hardware resources, simultaneously accessed by a large number of tenants running heterogeneous workloads. The number of tenants as well as their activity and resource needs dynamically change over time, and the cloud infrastructure reacts to this by reallocating existing or provisioning new resources. Besides metrics such as the number of tenants and overall resource utilization, performance testing in the cloud must be able to answer many more questions: How is the quality of service of a tenant impacted by the constantly changing activity of other tenants? How long does it take the cloud infrastructure to react to changes in demand, and what is the effect on tenants while it does so? How well are service level agreements met? What is the resource consumption of individual tenants? How can global performance metrics on application- and system-level in a distributed system be correlated to an individual tenant's perceived performance?

In this paper we present *CloudPerf*, a performance test framework specifically designed for distributed and dynamic multi-tenant environments, capable of answering all of the above questions, and more. CloudPerf consists of a distributed harness, a protocol-independent load generator and workload modeling framework, an extensible statistics framework with live-monitoring and post-analysis tools, interfaces for cloud deployment operations, and a rich set of both low-level as well as high-level workloads from different domains.

CCS Concepts

•**Computing methodologies** → **Simulation environments; Simulation tools;**

Keywords

Performance testing; workload modeling; load generation; statistics collection; multi-tenancy; cloud

1. INTRODUCTION

Cloud providers offer on-demand infrastructure, platform, or software as a service (abbreviated IaaS, PaaS, and SaaS) to their users, called *tenants*, on a subscription-based payment model. Rather than purchasing hardware and software up-front and integrating, deploying and managing it in their own data center, tenants rely on the cloud provider for these tasks. In order to minimize their cost, cloud providers will try to maximize the number of supported tenants on their infrastructure by sharing physical resources among tenants. This is typically achieved by some form of virtualization on the hardware, operating system, platform, or application layer. While providers aim at maximizing the *overall* performance of their infrastructure, tenants are more concerned about their *individual* performance. This can easily create a conflict of interest, as the provider needs to restrict a tenant's resource consumption in order to host as many tenants as possible, while tenants may need more resources in order to achieve their desired performance objectives. Provider and tenant therefore often negotiate a service level agreement (SLA), in which the provider guarantees a certain minimum quality of service (QoS), sometimes combined with penalty payments if the SLA is not met. As the provider has little control over the workloads of its tenants, it can become challenging to fulfill their SLAs as workload characteristics of tenants may change at any point in time. A tenant on the other hand may complain if it suddenly experiences performance degradation due to a noisy neighbor although its own workload did not change.

ICPE'17, April 22 - 26, 2017, L'Aquila, Italy

ACM ISBN 978-1-4503-4404-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3030207.3044530

Performance testing of cloud deployments is equally important for providers and tenants. Cloud service providers must be able to simulate realistic tenant behavior in order to optimize their infrastructure to best support a large number of tenants without violating their SLAs. Tenants must have certainty that the offered cloud services fulfill their requirements before moving their applications and workloads into the cloud. In order to support such tests, a performance test framework must be able to span potentially hundreds of nodes from which it collects data across all layers, including system-, platform- and application-level statistics. Rather than assuming a static infrastructure, it must support nodes to be added or removed from the system under test (SUT) at any time during a test run. To test the robustness of the infrastructure, it must be able to inject errors into the system to trigger application or node failures, and initiate operations such as the provisioning of new tenants or live-migration of a workload. It must be able to simulate the behavior of thousands of tenants running heterogeneous workloads, while controlling each tenant's workload individually to simulate their changes in load. Each tenant's throughput and response times must be measured separately to verify the fulfillment of SLAs. For metering and charge-back, it must also be possible to measure each tenant's resource consumption individually and correlate it to the tenant's behavior. Last but not least, such a framework must provide efficient ways to aggregate, post-process, report, analyze, and correlate the collected data across all nodes, layers, and tenants in configurable and comprehensive ways. While many existing performance test tools cover some of these requirements, we are not aware of any tool that addresses all of them.

In this paper we present *CloudPerf*, a performance test framework we have been developing at Oracle since 2012. The main design goals of CloudPerf were to create a distributed, dynamic, and extensible framework scalable to hundreds of nodes and thousands of tenants, with a protocol-independent load generator providing workload modeling abstractions for the implementation of arbitrary workloads, a statistics framework for live-monitoring, post-processing and analysis of workload and system statistics across all nodes, layers, and tenants, and integration with deployment operations in cloud environments. All necessary tasks from workload development, test deployment, test execution and statistics collection to test evaluation and analysis should be provided end-to-end under the umbrella of a single, tightly integrated framework.

The main contributions of this paper are:

- CloudPerf, a performance test framework for distributed and dynamic multi-tenant environments
- a detailed description of CloudPerf's architecture and components
- examples how CloudPerf can be used for performance testing of use-cases relevant to cloud deployments

The remainder of this paper is organized as follows: In section 2 we formulate requirements for cloud performance testing. In section 3 we describe the architecture and components of CloudPerf, and illustrate key features of CloudPerf in section 4. Related work is summarized in section 5, and we conclude with section 6.

2. REQUIREMENTS

Before designing and developing a performance test framework, requirements need to be formulated which the framework shall address. In this section, we gather requirements important for performance testing in cloud environments. This requirement list is certainly not exhaustive, but covers many central and relevant aspects and incorporates as well as exceeds requirements formulated by other researchers [7][10][11]. In sections 3 and 4, we then discuss how CloudPerf fulfills these requirements.

R 1 (SCALABILITY). *Scalability to hundreds of nodes, thousands of tenants, days of test duration, and virtually unlimited number of end users, transaction rates, and concurrent sessions.*

R 2 (ELASTICITY). *Support of elastic SUTs with nodes being added and removed from the SUT during a test run.*

R 3 (AVAILABILITY). *Support of destructive tests and accidental node deaths during a test.*

R 4 (GENERIC WORKLOAD MODELING). *Modeling abstractions for the implementation of arbitrary high-level as well as synthetic workloads using reusable building blocks.*

R 5 (PROTOCOL INDEPENDENCY). *Load generation independent of any protocol or application domain to allow implementation of workloads for any domain using the same framework and modeling abstractions.*

R 6 (DYNAMIC LOAD GENERATION). *Support of open and closed system load generation models for synthetic load generation with dynamic change of request rates, concurrency, and other workload parameters.*

R 7 (LOAD REPLAY). *Replay of captured workload profiles at configurable speed.*

R 8 (MULTI-TENANCY). *Support for mixed workloads and individual per-tenant control of any load generation or workload parameter.*

R 9 (FAULT INJECTION). *Injection of faults and execution of arbitrary actions during a test run to simulate or trigger errors and other operations.*

R 10 (CLOUD LIFE CYCLE OPERATIONS). *Triggering and measurement of deployment and other cloud management operations performed by the cloud provider such as provisioning or live-migration of tenants.*

R 11 (STATISTICS COLLECTION). *Collection of fine-grained load generation, system- and application-level statistics across all nodes and layers.*

R 12 (CONDITIONAL EVENTS). *Conditional execution of load changes, fault injection or management operations based on observed tenant or system behavior.*

R 13 (METRICS AND RESULTS). *Calculation of metrics and result verification using arbitrary statistics.*

R 14 (LIVE OBSERVABILITY AND INTERACTION). *Live observability of a test run including monitoring of any collected statistics during runtime as well as interactive intervention (such as change of load) during a run.*

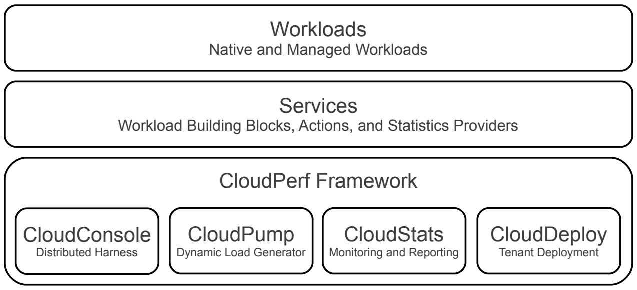


Figure 1: CloudPerf Framework Architecture

R 15 (Time-based Statistics Aggregation). *Aggregation of statistics based on time intervals defined by load changes or other triggers to allow evaluation of statistics, metrics, and results across the entire test run as well as for each individual interval.*

R 16 (Flexible Reporting). *Flexible report generation of statistics, metrics, and results using user-defined templates supporting arbitrary aggregation and correlation of statistics from any source. Report generation across multiple test runs for test comparison.*

R 17 (Interactive Analysis). *Tools for interactive analysis and correlation of all collected statistics for troubleshooting and deep-dive analysis.*

R 18 (Repeatability and Automation). *Repeatable test execution using pre-configured events and interfaces for automated test submission.*

R 19 (Extensibility and Reusability). *Well-defined APIs for high degree of reusability and extensibility, especially for statistics collection, workload development, and fault injection.*

R 20 (Ease of Use). *Ease of use from workload development, test deployment, test execution, to test evaluation.*

3. ARCHITECTURE

Key objectives of CloudPerf were to design a scalable, extensible, flexible, and reusable framework for performance tests of all kinds, that provides as much as possible common functionality in the framework without limiting its use to any specific domain. In order to accomplish this, we have designed CloudPerf in a very generic way using abstractions that are common across nearly all workloads, statistics, or deployment types, and where necessary also allow to by-pass framework components to even cover those corner-cases not directly supported by our framework. CloudPerf is written in Java and therefore runs on any environment supported by Java.

CloudPerf is a modular software product comprised of individually developed and packaged *modules*. The CloudPerf *Framework* module consists of four core components, *CloudConsole*, *CloudPump*, *CloudStats*, and *CloudDeploy* complemented by a set of tools. The framework module is extended through *Service* and *Workload* modules (figure 1). The remainder of this sections describes the framework components, services, and workloads in detail.

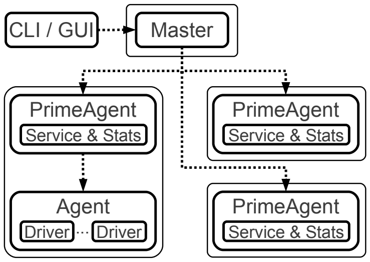


Figure 2: CloudPerf Process Architecture

3.1 CloudConsole: Distributed Harness

CloudConsole is a distributed harness which manages the overall test deployment, configuration, submission of test runs and events, communication between nodes, result gathering, and live polling of statistics.

3.1.1 Deployment and Process Model

A CloudPerf deployment is managed through a *Master* process running on a dedicated node, which serves as the single point of administration. From this node, the CloudPerf software is automatically installed or updated on all configured nodes of the test deployment. On each node of an installation, a *Prime Agent* process is started. The prime agent remains idle until being requested to participate in a run by the master. A test run may be submitted to all or just a subset of the provisioned prime agents, separating software distribution from test execution. Prime agents serve two main purposes (also simultaneously) during a test run, the collection of system statistics and the management of load generators. While statistics collection is done by the prime agents themselves, they spawn separate *Agent* processes to run the actual load generators (*Driver*) on selected nodes. A sample deployment is illustrated in figure 2.

Note that CloudPerf does not explicitly distinguish clients and SUT, but treats all nodes the same. Whether a node that runs a load generator is considered client or SUT is entirely the decision of the performance tester. CloudPerf also allows to run master, prime agent, and agents all on the same node for single-node deployments, and even supports to run them inside a single JVM for development purposes.

During a run, the master supervises all participating prime agents and polls statistics from them, which in turn supervise and poll statistics from all agents they have spawned. The inter-process communication is implemented through Java Remote Method Invocation using parallel threads to ensure scalability to hundreds of nodes (R1). Users can connect to the master using graphical (GUI) and command-line (CLI) tools to monitor and control test runs.

3.1.2 Run Configuration and Submission

A run is configured through a set of XML-based configuration files. The CloudPerf framework itself provides *framework* and *cloudpump* configuration files which specify framework parameters such as ports, JVM options, trace and debug settings, and basic load generator parameters. *Service developers* create *service* descriptions specifying resources offered by a service as well as reporting of service-provided statistics. *Workload developers* create *workload* descriptions specifying Java class names, transaction names and distributions, metrics, result checks, and arbitrary other parameters offered by workloads. A *deployment description* is created by a *performance tester*, who specifies the nodes to be used for a test run, the statistics providers to run, their configuration, the assignment of load generators to hosts, as well as workload-specific deployment information such as URLs of web servers or databases for the load generators to connect to. Wildcards allow prime agents to join a test run after run submission, for example if new nodes are to be dynamically created during a run (R2). Additionally, the performance tester creates a *run description*, in which the test flow is specified in form of load change events and actions, their timing and load parameters such as injection rates or concurrency. Finally, a *report template*, which is provided by the framework but may be customized by the performance tester, specifies the content of the generated performance report.

Configuration parameters can be overwritten by configuration files in the order described in this section, which for example allows a performance tester to override basic load generator or workload parameters in the deployment or run descriptions without modifying the load generator or workload configuration files. Except for the basic framework configuration, which is read by each prime agent during startup, all other configuration files are passed to the master through a GUI or CLI when submitting a test run. These files can therefore be edited in a single, centralized location. Performance testers can use a web-based user-interface (WebGUI) to generate deployment and run description files automatically rather than manually editing them (R20).

3.1.3 Events

Events can trigger load changes in load generators (see section 3.2) or execute *actions* inside prime agents. Actions are small pieces of Java code provided by services which allow the execution of arbitrary code on any host at any point in time. With actions it is possible to inject faults into the SUT by killing a process, panicking a host, or simulating memory shortages by consuming all available memory (R9). Actions can also trigger operations such as the creation of new VMs, adding of CPUs to a VM, or the migration of a tenant to another host (R10).

Events can be pre-configured in the run description, depend on the outcome of actions, triggered by conditions based on the evaluation of statistics (R12), or interactively submitted through GUIs and CLIs (R14).

3.1.4 Result Gathering

In order to minimize network bandwidth consumed by the harness, all captured statistics are kept locally on each node during runtime. At the end of a test run, each prime agent and agent perform a local post-processing of their statistics in parallel (R1) before sending them to the master. Once the master has received all statistics, it invokes the reporter for report generation.

3.1.5 Live Polling

Any collected statistics such as transaction throughput and response times or CPU utilization can be polled live during runtime. Clients like LiveView or CLIs (see section 3.7) can connect to the master during runtime to fetch and display live statistics from all nodes. This allows performance testers to monitor a test while it is running (R14) rather than waiting for a report to be generated after test completion.

3.2 CloudPump: Dynamic Load Generator

The fundamental concepts of load generation are entirely independent of the type of workload, yet workload developers often not just develop a workload, but also the load generation capability to execute it, and thus reinvent the wheel with every new tool. This often leads to tools with only very restricted workload generation capabilities (see section 5).

With CloudPerf we take a different approach and provide all load generation capability inside the framework decoupled from the individual workloads. Workload developers only need to implement the business logic of their workload, while all load generation functionality is provided by the framework. This not only simplifies workload implementation as developers can focus on the business logic, but also provides consistent and reusable load generation capabilities with identical characteristics across all workloads (R19). To achieve this, we have designed a workload implementation model providing abstractions common across all workloads (R4). The scheduling of requests is independent of the workload domain and communication protocols (R5) and supports dynamic load changes (R6), load replay (R7) as well as multi-tenancy with per-tenant load control and reporting (R8).

3.2.1 Workload Modeling

CloudPerf provides a generalized workload model through APIs and abstract classes. Workload and service developers use these modeling concepts to implement workloads and services. This section briefly describes the core concepts of our workload model.

Implemented by Service

Operation A reusable building block to perform a certain task (examples: HTTP GET, database prepared statement, network read, disk write)

Connection A communication channel on which to run an operation (examples: HTTP connection, JDBC connection, TCP connection)

Connection Pool A pool of connections or a connection factory to be used by clients to connect to a server

Connection Listener A listening endpoint of a server for incoming client connections

Implemented by Workload

Transaction A set of operations glued together by business logic (examples: add to shopping cart, place a new order, receive and reply)

Session A sequence of related transactions (scheduled with think or cycle time) executed in the same context on behalf of a common user (identified through a common user ID)

Context The state of a session

Load A vector of values describing a workload's magnitude and parameters (such as rate, concurrency, user range, and custom attributes)

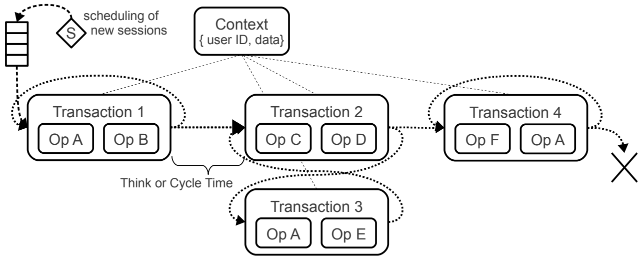


Figure 3: CloudPerf Workload Modeling

Figure 3 illustrates a workload consisting of four transactions. In this example, a new session always begins with *Transaction 1* (initial transaction), which is scheduled by the framework's scheduler *(S)* according to a configured injection rate or concurrency (see section 3.2.2). The framework also creates a session context for this session and picks a user ID based on a configurable distribution. *Transaction 1* executes two operations *Op A* and *Op B* provided by a service, and then selects a subsequent transaction to run for this session and instructs the framework to schedule either *Transaction 2* or *Transaction 1* next. The scheduling of subsequent transactions may depend on the outcome of the current transaction or any other criteria and uses either think or cycle time to determine when to run the subsequent transaction. *Transaction 2* participates in the same session and therefore references the same session context through which data can be shared across transactions of the same session. Further state transitions eventually execute *Transaction 3* and *Transaction 4*, which may re-schedule itself or terminate the session.

If this example was a workload simulating the use of an online shop, *Transaction 1* could be a login transaction in which a user tries to login to their account, which is repeated until the login was successful. The user name could be derived from the user ID selected by the framework. *Transaction 2* could be a transaction in which the user browses a catalog of shopping items. In *Transaction 3*, the user may add an item to their shopping cart, and then continue browsing for more items in *Transaction 2*, until the user finally proceeds to *Transaction 4* in which the checkout and payment could be implemented. After successful payment, the session is completed. Operations could be HTTP GET and POST operations to load web pages from a web server and submit user input if this was a web workload, or prepared statements to be run against a database if this was a database workload. Any metadata such as web cookies, number of login attempts, or items already placed into the shopping cart, would be stored in the session context.

Figure 4 illustrates the use of connections. The first model depicts a classical connection pooling model in which connections are borrowed from a pool at the beginning of each transaction and returned back to the pool at the end of a transaction. In the second model, the connection pool is used as a factory: A new connection is created in the first transaction of the session and then stored in the session context. Subsequent transactions of this session use the previously created connection from the context, and the last transaction of the session closes the connection. CloudPerf makes no restrictions as to how many connections a session may create or borrow (as long as sufficient connections are available in the pool or the pool has not yet reached its maximum size). Also arbitrary combinations of the illustrated models are possible. For example, a session may store a borrowed connection in its session context or use both borrowed and newly created connections in the same session.

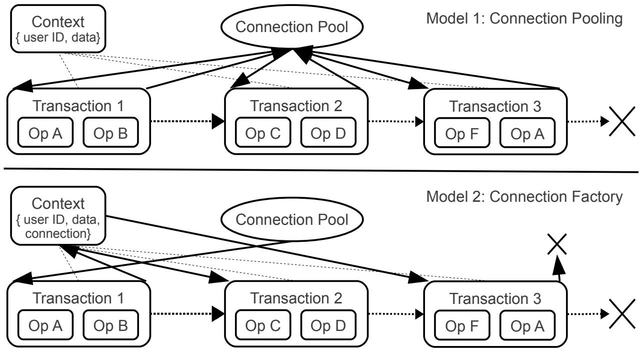


Figure 4: CloudPerf Connection Handling

This model has proven to be very flexible for implementing any arbitrary workload. During workload design, developers will try to map a workload's elements onto the provided modeling abstractions in the most elegant way. Depending on the workload, some of these abstractions might not be applicable and will consequently not be used. For example, disk I/O workloads have no concept of connections. Other workloads consist only of unrelated events and do not require sessions. In those cases, operations may perform disk reads and writes without using connections, and sessions may consist of only a single transaction. User IDs chosen by the framework in uniform or non-uniform ways typically identify a range of data accessed by a workload (for example customer records in a database). A disk I/O workload without the concepts of users may instead use the user ID to decide the offset in a file for a read or write operation.

3.2.2 Load Generation

Load generation is implemented inside a *driver* provided by the CloudPerf framework. In a multi-tenant deployment, each individual tenant will typically be simulated by a dedicated driver. Mixed workloads will typically be run with one driver per workload and tenant. While drivers are highly performant[1], scale-out by instantiating multiple drivers per tenant provides virtually unlimited scalability for load generation (R1).

Figure 5 illustrates the main components of a CloudPerf driver. A *Scheduler Thread* creates *jobs* for new sessions at either a configurable injection rate (open system model) or based on a target rate of concurrent sessions to maintain (closed system model), and enqueues these jobs in a *Scheduling Queue.* An idle *Driver Thread* from a pool of threads dequeues the next job. If the job requires the start of a new session, it picks a random initial transaction from a transaction distribution table, creates a new session context, and executes the business logic of that transaction. If the transaction does not schedule a subsequent transaction, the driver

[1] A single driver can reliably schedule about 200,000 new sessions per second on Intel Westmere servers while just consuming a small fraction of CPU. Overall CPU consumption of a driver primarily depends on the implemented workload business logic.

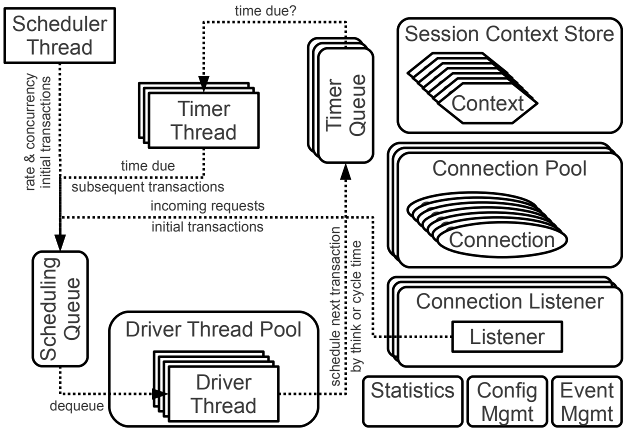


Figure 5: CloudPerf Driver Architecture

thread terminates the session after executing the transaction and deletes the session context. If the transaction has requested a subsequent transaction to be scheduled, the driver thread computes the due time of the subsequent transaction based on think or cycle time and inserts a job for the subsequent transaction into a *Timer Queue*[2]. For each timer queue, a *Timer Thread* dequeues jobs that have reached their scheduled time, and inserts them into the scheduling queue, where an idle driver thread will dequeue them and execute the scheduled subsequent transaction.

Note that subsequent transactions of the same session may execute on different driver threads and must therefore use the session context rather than thread-local variables to store session-related data. Driver threads are only used during execution of a transaction and then returned back to the pool. With this design, CloudPerf supports a virtually infinite number of concurrent sessions (only limited by the memory needs of the session contexts) where the number of driver threads only depends on the transaction rate and duration but not the think or cycle time or the number of concurrent sessions (R1).

The use of a scheduling queue decouples load injection from load processing and provides an open system model that simulates request-driven workloads much more accurately than closed system models[16]. Note that the scheduling queue only serves as a buffer and will be mostly empty if load processing keeps up with the injection rate. CloudPerf provides multiple options to handle overload, including the discarding of jobs when the scheduling queue is full and limits on the maximum queueing time for jobs in the queue (those jobs will be counted as failed transactions), which maintains an open system model also during overload.

Transactions may use connection pools provided by the driver as described in section 3.2.1. The driver maintains a configurable number of connections in each pool, specified through minimum and maximum size, and also supports resize operations during runtime, recreation of connections at configurable rates, and other features.

The driver captures statistics about transaction and operation throughput, success and failure rates, and response times including percentiles as well as resource usage of internal resources such as connection pool, driver thread pool, and scheduling queue length. User-defined timers and statistics counters can be used by service and workload developers for additional timing and statistics. Through APIs the driver provides developers with access to workload and other configuration parameters and reacts to events it receives from the master, which may cause it to change injection rates or target concurrency for the scheduler, resize a connection pool, change a workload parameter, or stop the workload.

For most workloads such as database and mid-tier workloads or CPU and storage micro-benchmarks, the driver will act as an initiator of requests. Through its *Connection Listener*, it can also act as a server, for example to implement client-server network micro-benchmarks. If the connection listener receives an incoming request, it will create a job for a new session and insert it into the scheduling queue, from which a driver thread will dequeue and execute it.

For synthetic workloads, requests are typically randomly generated based on a configurable distribution, injection rate, and concurrency. A random selection however, even if following a certain distribution, has limits and may not always be an accurate simulation of a given scenario. For example, a storage and its caching strategies may react differently to a synthetic random read/write mix than to the I/O profile of a real application, which generates specific sequences of read and write requests of a certain size on particular file descriptors. CloudPerf therefore also supports the replay of a previously captured profile in which the timing, type, and parameters of transactions are described in a text file. During load replay, the scheduler thread reads this profile and creates jobs for corresponding transactions accordingly. A configurable replay speed allows users to execute a profile faster or slower than it was captured (R7).

3.2.3 Load Changes and Dynamic Load

Load parameters can be changed at any time during a test run through load change events that are delivered to all or a select subset of drivers (R6). Drivers can increase or decrease injection rates, concurrency of sessions, replay speed, connection pool sizes, or any other workload parameters upon load change events. Changes in value can either be immediately or gradually applied over a configurable period of time.

To simulate constantly changing injection rates, CloudPerf also supports to configure load curves that continually adjust the injection rate based on a mathematical function with a certain period and magnitude such as a sine or gaussian function.

3.2.4 Mixed Workloads and Multi-Tenancy

Since two drivers share no resources (other than potentially the JVM or host they are deployed on), they act completely autonomously and can simulate individual tenants or workloads (R8). Each driver can have dedicated configuration and load parameters, run different workloads or transactions, and keeps its individual statistics to measure throughput and response times.

CloudPerf allows users to deploy an arbitrary number of drivers in an agent process, an arbitrary number of agent processes per host, and distribute agents across an arbitrary number of hosts. These deployment options enable Cloud-

[2] For better scalability, each driver maintains a set of timer queues, which are used in a round-robin fashion. Subsequent transactions with a think-time of 0 bypass the timer queues and are directly re-executed in the same driver thread.

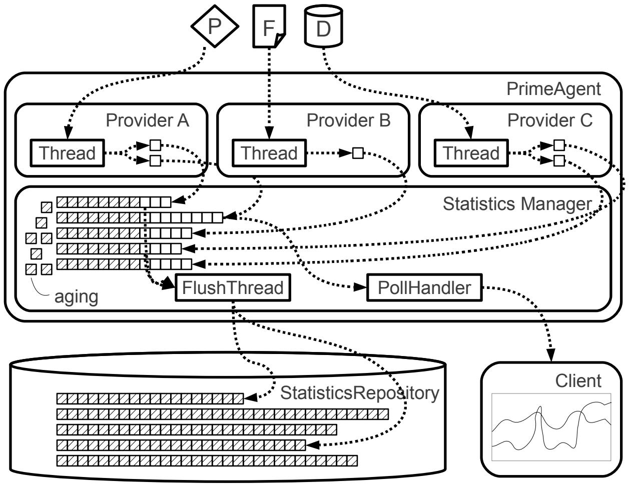


Figure 6: CloudPerf Statistics Framework

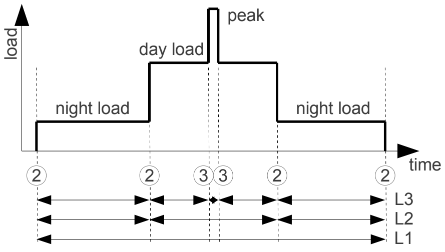


Figure 7: Time-Based Statistics Aggregation

Perf to scale to thousands of individually controlled and reported tenants running same or different workloads (R1).

3.3 CloudStats: Monitoring and Reporting

Monitoring and reporting are essential tasks of performance engineering and performance testing work. Most performance test tools are capable of capturing throughput and response time statistics for the load they generate, yet only few also capture comprehensive system-level or application statistics. The use of dedicated monitoring tools leaves the performance engineer with the task to integrate load generation and monitoring, which becomes increasingly challenging if load generation is not steady and load levels are changed throughout the test so that load changes need to be correlated in time with system statistics.

CloudPerf therefore natively integrates monitoring (R11) and reporting (R16) by providing an extensible and generic statistics framework (R19). This framework consists of a statistics repository in which all statistics are stored in a uniform way, and a set of pluggable statistics providers that capture data and add them into the repository, complemented by a GUI for live-monitoring, a customizable reporter for report generation, and other post-processing tools.

3.3.1 Data Organization and Repository

CloudPerf stores statistics in form of *series* which each consist of a sequence of *samples*. Each sample is a time-value pair. Series are created by statistics *providers* in a hierarchical namespace. The fully qualified name of a series consists of the provider name (*hostname*, *service*, *providername*, *providerid*) and the series name (*category*, *subcategory*, *series*).

Figure 6 illustrates the statistics framework architecture. Statistics providers are implemented by services and started on a host inside a prime agent. They register their series with the *statistics manager* which maintains the statistics repository and periodically flushes data to disk as it is being captured. Flushing also ensures that statistics data is not lost during destructive tests where nodes are crashed or rebooted during runs (R3). Recent samples are kept in memory to allow clients to poll for selective statistics, while old samples are aged out and purged from the statistic manager's cache to save memory. Statistics providers may use arbitrary ways to obtain their data such as the forking of a process (P) and reading of its output, tailing of a log file (F), connecting to an application like a database (D) and querying performance statistics, sampling of counters or registering for notifications. They may add samples to their series at arbitrary times and intervals.

3.3.2 Live Observability

Performance testing is usually an iterative process in which bottlenecks are eliminated and parameters adjusted until a workload or system behaves as desired. To facilitate such process better, CloudPerf provides a *LiveView* GUI for live observability of the system during runtime to allow performance testers to troubleshoot performance during the run rather than waiting for a report to be generated at the end (R14). LiveView uses CloudPerf's polling interface on the master to register for statistics series of interest and periodically poll latest samples. The master forwards these requests to all prime agents, which in turn forward them to all agents, and then aggregates the results before returning them back to the client. With statistics polling, any statistic collected on any node in a distributed environment can be observed and displayed live during runtime.

3.3.3 Conditions and Alerts

Conditions are rules based on mathematical expressions configured by a user for a certain set of statistics series on a node, which are evaluated by the statistics manager at runtime. When the configured expression evaluates to *true*, the statistics manager raises an alert, which triggers an action (delivered through an event) on any arbitrary host (R12). Performance testers can use this mechanism in many ways, for example to add VCPUs to a virtual machine if CPU utilization exceeds a threshold or to adjust load generation parameters based on the observed resource usage or response times.

3.3.4 Time-Based Statistics Aggregation

When load is dynamically changed throughout a test run, any statistical summary that only considers the run as a whole but does not distinguish its individual phases is of little value. Figure 7 illustrates a test run consisting of three major phases: a night-load, day-load, and night-load phase. During the day-load phase, a spike in traffic was simulated. Load change events in CloudPerf are tagged with a *level*, shown as a circled number in this figure. A level 1 event is automatically inserted at the very start and end of a test. Event timestamps determine the aggregation intervals for all captured statistics. Each aggregation interval is defined by two events of same or lower level. In the example, an L1 aggregation would represent statistics for the entire tests, while the L2 aggregation would distinguish night-load, day-load, and night-load phases, and the L3 aggregation would further break the day-load phase into the time period before, during, and after the peak. Statistic samples for all series are aggregated for each of these intervals (R15).

3.3.5 Customizable Reporting

At the end of a test run, all statistics captured by prime agents and agents are copied to the master's repository. The CloudPerf *reporter* then creates an HTML report for the test run. The content and format of the report is controlled through a customizable XML template (R16). It defines how statistics should be aggregated (for example, total throughput as the sum of the throughput of each individual driver, or disk servicing times as the average of all servicing times of disks of a particular host) and reported (in tabular form for each of the aggregation intervals, or plotted as a graph). The reporter makes no restrictions as to which series to aggregate or report and correlate in a common table or graph. Further report contents include captured system and configuration information, log files, and links to raw data. While the default report template satisfies most needs, users are encouraged to adapt it to fine-tune the reporting of their results to organize and summarize data in the most suitable form for their experiments.

The reporter can be manually rerun for previous tests to regenerate reports with different templates. Performance engineers often run series of tests in which selective parameters are varied to study their effects. To facilitate such analysis, the reporter can compare up to ten tests and create a comparative report in which data for multiple tests is reported side-by-side including relative differences of values.

Workload developers and performance testers can further specify *metrics* and *result checks* to compute higher-level metrics from the captured statistics or validate their values to determine whether a test should be considered successful or failed (R13). Metrics and result checks are either defined in XML or implemented as Java-callbacks for more complex evaluation. Services may also implement Key Performance Indicators (KPIs) which classify certain statistics or metrics into user-defined categories such as low, medium, and high.

3.4 CloudDeploy: Deployment Operations

Section 3.2 discussed the generation of user load. While the load profiles of users (or tenants) play a central role in cloud deployments, they are not the only activities that determine the overall performance of an environment. Administrative operations performed by the cloud service provider such as tenant deployment operations are equally important. CloudPerf provides the following abstractions for tenant deployment and management operations (R10):

Create Creation of a virtual resource

Start Starting of a virtual resource

Modify Modification of a virtual resource

Migrate Migration of a virtual resource

Stop Stopping of a virtual resource

Destroy Deletion of a virtual resource

Virtual resources describe the owning tenant, resource type and identifier, as well as arbitrary metadata. These abstractions are implemented by services which integrate them with the underlying cloud infrastructure. For example, a service for virtual machines could implement these abstractions as creation, boot, adding and deletion of vCPUs, migration, shutdown, and deletion of virtual machines. A service for Oracle Pluggable Databases (PDBs) could implement them as cloning, opening, configuration of resource management plans, migration, closing, and dropping of PDBs.

Deployment operations in CloudPerf are a special form of actions that can be triggered at runtime through events directed to any set of hosts. The framework will record their rate as well as start, end, and execution times to correlate deployment operations with performance metrics such as network bandwidth usage during VM creation or quality of service during live-migration for both the migrated as well as all other tenants.

3.5 Services

The CloudPerf framework is designed to be entirely agnostic of any particular product or domain. Services developed and packaged as individual modules adapt, map, and extend the framework capabilities to particular products and domains. They implement well-defined APIs and are relatively easy to develop (R19). Services provide:

Reusable workload building blocks such as connection pools and listeners, connections, and operations

Actions for performing deployment operations and other arbitrary tasks at runtime

Statistics providers for capturing of runtime statistics

Configuration gathering for capturing of system parameters

Precondition checks for validating a system configuration before a test run

Among the services we have developed are *HTTP*, *JDBC*, and *Network* services to provide the necessary building blocks for middleware, database, and network workloads. *Solaris* and *Linux* services provide OS-level statistics ranging from CPU utilization, memory usage, network and disk activity, and process statistics to hardware performance counters, NUMA and resource management statistics, and many more. A *GenericOS* service is capable of running generic scripts or tailing arbitrary log files and extracting data from them through configurable regular expressions. *Oracle*, *WebLogic* and *HotSpot* services capture detailed database, application server, and JVM statistics, and further services implement deployment operations for VMs and PDBs.

3.6 Workloads

As discussed before, the CloudPerf framework only provides generic modeling abstractions for workloads to implement workloads of any domain. The effort for the implementation of a new workload depends largely on the complexity of the workload and is often fairly low for simple workloads.

We have implemented numerous workloads ranging from high-level workloads to micro-benchmarks, and continue to add new ones. Our workloads include database OLTP workloads such as TPC-C, flight booking and the customizable

CRUD workload[14], DSS and analytical workloads such as a generic customizable DSS workload capable of running arbitrary queries and query streams and HTTP-based middleware workloads. Networking, disk, and scheduling workload provide powerful micro-benchmarks for multi-node and multi-tier networking tests, storage benchmarking, and optimization of operating system scheduling.

A generic *wrapper workload* is capable of wrapping existing workloads or commands to integrate them into CloudPerf and run them through the framework in a distributed environment without the need of porting or rewriting them. The concurrency of running commands can be changed any time through load change events, and regular expressions help to extract data from the output of these commands to feed it back into the statistics framework.

3.7 Tools

The CloudPerf framework includes several tools, many of which have been mentioned in previous sections already. The most important tools are:

Installer to install, update, and deinstall modules

DataLoader to populate a database, flat file, or other medium with test data

CLIs to restart master and prime agent processes, submit workloads and events, query the status of scheduled runs, and automate test submission (R18)

WebGUI to create deployment and run configuration files, submit workloads, and browse through results

LiveView to submit workloads, interactively change load, and monitor statistics during runtime (figure 9)

Reporter to create reports and compare test runs

StatsCollector for stand-alone statistics collection

StatsTool to export statistics from our repository for post-processing in spreadsheets or other tools

Performance Data Analyzer (PDA)[4] is not included but tightly integrated with the framework and supports interactive analysis and correlation of statistics (R17).

4. CLOUD PERFORMANCE TESTING

Section 3 described the architecture and key features of CloudPerf. In this section, we will illustrate on some examples how these features can be leveraged for performance testing in the cloud or other dynamic and distributed environments.

4.1 Test Series

In performance engineering it is very common to run test series in which a single parameter is changed from run to run to study its effect. Arguably most common are scaling tests in which either the injection rate or concurrency are increased. Though not directly related to cloud, such kind of experiments are greatly simplified by CloudPerf's ability to change load during a test. Rather than running a series of individual tests, CloudPerf enables performance engineers to run the entire test series in a single test run with multiple individually reported load steps where injection rate or concurrency are adjusted through load change events. This not only speeds-up test execution as lengthy ramp-up phases, pre- or post-processing steps can be avoided or greatly reduced, but also reduces the risk of accidental changes to the environment in between tests. Also workload parameters that alter the behavior of the workload can be changed through load change events. System and application parameters can be altered during a test through action events such as the *exec* action implemented in the Solaris and Linux services which executes any arbitrary command.

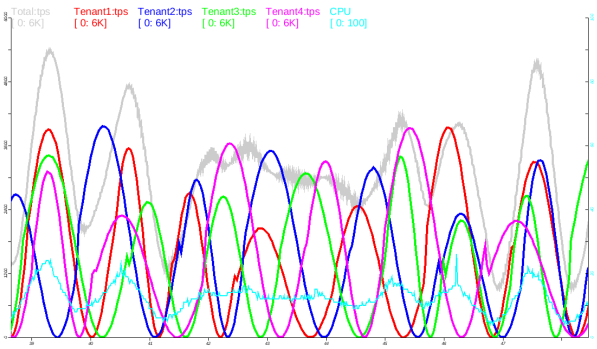


Figure 8: Dynamic Load Curves

4.2 Dynamic Load Curves

Performance at *steady-state* was and is widely used by performance testers to assess and report performance of an environment, yet it is a simplification which neglects the elastic aspects of shared deployments such as cloud. As infrastructure is shared among workloads or tenants, the elasticity of an environment - its ability to react to changes in demand - becomes an essential factor of its overall performance characteristics. Especially as cost considerations drive service providers to a higher degree of sharing, the risk of resource contention increases if tenants consume more resources than anticipated, be it network bandwidth, storage IOPS, or CPU cycles. The simulation of dynamic load curves is therefore a crucial feature of performance test frameworks for the cloud.

Figure 7 already illustrated how CloudPerf can be used to simulate different levels of activity such as day- or night-load phases as well as sudden load peaks through load change events. As discussed in section 3.2.3, tenants simulated by individual drivers can be controlled separately by directing load change events to selective drivers, creating potentially different load patters for each tenant.

Load change events however become impractical if load for a large number of tenants shall be constantly and individually adjusted. With the support of mathematical functions to generate per-tenant load curves with randomized period and magnitude, dynamic load patterns can be easily configured in CloudPerf even for hundreds of tenants (R6). Figure 8 shows a plot generated by CloudPerf for randomized sine load curves of four tenants.

4.3 Noisy Neighbors

Load change events and load curves are also ideal instruments to study the effects of noisy neighbors and the potential degradation of quality of service due to interference with other tenants. Such experiments can be conducted by keeping the injection rate for a primary tenant constant while increasing the rate of colocated tenants (neighbors) through load change events. The noisiness of a neighbor typically depends not just on its activity, but also its proximity to the primary tenant. The experiment can be further enhanced by changing the proximity of colocated tenants through action events, for example by changing the CPU binding of their virtual machines or containers from a different socket to the same socket or even same cores as the primary tenant. Throughput and response times of the primary tenant can then be correlated to other system metrics such as cache misses and stalls (also captured by CloudPerf) to assess their effect on the quality of service on the primary tenant.

Note that for request-driven workloads such experiments are much more meaningful with an open system load generation model than with a closed one. In an open system model, a fixed injection rate for the primary tenant will maintain its throughput as long as no resource is saturated even though its quality of service degrades. This degradation appears in form of increased response times, for example due to higher network, storage latencies, or cache misses as in the above sketched experiment, which can be directly compared to identical throughput of the tenant in isolation. In a closed system model, the quality of service degradation will manifest itself in a reduction in throughput for the primary tenant. In other words, the primary tenant would appear to be changing its behavior due to its noisy neighbors, which is typically not an adequate simulation of request-driven workloads where requests originate from external end-users and are fairly independent of response times unless they exceed critical thresholds.

4.4 Tenant Deployment and Node Addition

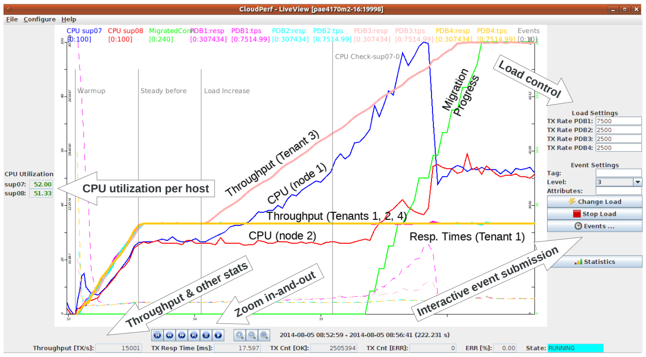
Not just the load of tenants constantly changes in cloud deployments, but also the number of hosted tenants. The time needed to provision a new tenant, the resource consumption of tenant provisioning, and the impact of tenant provisioning on the quality of service of other tenants are important performance attributes of cloud environments. As discussed in section 3.4, deployment operations in CloudPerf are implemented by services and may invoke cloud management software APIs to deploy or destroy tenants. At runtime, they are triggered through deployment events. Performance testers may use this to evaluate the cost of tenant deployment or the scalability of an environment as the number of tenants increases.

If new virtual machines are created for a tenant, a prime agent in that VM can be automatically started when the VM boots. It will then contact the master, which will lookup that prime agent's *role* and assign it a task, for example to collect statistics or start load generators. This way newly started hosts can join a running performance test and participate as if they had been there from the beginning, supporting dynamic changes of the SUT at runtime (R2).

4.5 Fault Injection and High Availability

Service availability is a key attribute of cloud offerings and is largely impacted by the robustness and fault tolerance of an environment as well as its ability and time needed to recover from failures. Stress testing in which faults are simulated is therefore besides performance testing another cornerstone of non-functional testing of cloud infrastructures. CloudPerf supports stress testing in many ways (R3).

Fault injection is enabled through actions, which execute arbitrary Java code and can be triggered on any host at any point in time through action events. Actions allow stress testers to kill processes, panic a host, change routing tables, offline CPUs, simulate memory shortages through allocation of large amounts of memory, and anything else that can be implemented in code.



CPU Utilization
sup07: 52.00
sup08: 51.33

Load Settings
TX Rate PDB1: 7500
TX Rate PDB2: 2500
TX Rate PDB3: 2500
TX Rate PDB4: 2500

Event Settings
Tag:
Level: 3
Attributes:

2014-08-05 08:52:59 - 2014-08-05 08:56:41 (222.231 s)

Throughput [TX/s]: 15001 TX Resp Time [ms]: 17.597 TX Cnt [OK]: 2505394 TX Cnt [ERR]: 0 ERR [%]: 0.00 State: RUNNING

Figure 9: LiveView: Live Monitoring and Control

The open system load generation model of CloudPerf decouples load injection from its processing. If processing stalls, the driver's internal scheduling queue will run full. For stress testing both queue length as well as maximum queueing time can be limited so that requests are discarded if not handled within a specified time. CloudPerf reports both the duration of total service downtime (no single request serviced successfully) as well as request failure rates. Together with other statistics such as throughput, response times, and system statistics, those are reported separately for the period before the service failure (the time of the action event), during service downtime or degradation until the service is restored (identified through either conditions or other watchdogs that raise alerts upon restoration), and after full restoration of the service.

If services provide information about the state of connections, the framework's connection pool will automatically discard and re-establish failed connections. For database stress testing, CloudPerf's *JDBC* service also integrates the *Oracle Universal Connection Pool (UCP)* for advanced failover strategies of JDBC connections.

CloudPerf is also robust against failures of any monitored host on which a prime agent is running. Captured statistics are continuously flushed to disk to minimize data loss to a few seconds. On a rebooted host, a prime agent can resume statistics collection and re-join an existing test run.

4.6 Cloud Life Cycle Management

Conditions as described in section 3.3.3 can be used to monitor statistics values and trigger actions upon certain criteria, for example to add VCPUs to an overloaded VM or migrate a tenant off an overloaded host. Such experiments can be useful to design or prototype cloud management software decisions or improved migration algorithms.

In [14] we presented and evaluated a new methodology to migrate PDBs between nodes of a cluster which we tested and prototyped using CloudPerf. Figure 9 shows a screenshot of CloudPerf's LiveView GUI visualizing throughput and response times of four tenants as well as CPU utilization of two hosts during the runtime of a similar live migration experiment. Tenants 1 and 3 are deployed on node 1, while tenants 2 and 4 are deployed on node 2. After a warmup and steady-state phase, the load of tenant 3 is gradually increased through a load change event, slowly driving up CPU utilization on node 1 (sup07). A condition has been configured on node 1 to monitor CPU utilization and trigger an action to initiate live-migration of tenant 1 off the overloaded node once it exceeds a certain value. The screenshot shows response times of all four tenants as dashed lines, clearly indicating the response time increase of tenant 1 as it is being migrated from node 1 to node 2, as well as tenant 3 which suffers from high CPU utilization on node 1. Once tenant 1 has been migrated off the overloaded node 1, the CPU utilization for that node drops and utilization on both nodes is balanced. The bold horizontal overlayed curves in the middle of the graph show throughput for tenants 1, 2, and 4. Throughput for the migrated tenant 1 is steady throughout the migration without any downtime.

CloudPerf not only provides the necessary features for such complex experiments, but can even be used to prototype improved algorithms or to monitor system behavior live during test execution.

4.7 Multi-Tenancy

In [17] we used CloudPerf to compare the efficiency and scalability of two different database deployment models by deploying and simulating 252 tenants. Tenants were split into three groups (small, medium, large) of 84 tenants each with different database sizes and injection rates. The focus of our study was to determine the resource needs and resulting throughput and response times for tenants in both deployments. Using CloudPerf's services providing operating system and database observability, we captured CPU consumption, memory footprint, storage I/O, and database statistics such as waits and buffer cache misses per tenant group for each deployment. In another experiment, we deployed application server and database clusters, storage infrastructure and load generators for 29 tenants across a total of 128 nodes (R8).

The ability of CloudPerf to control this many tenants on a distributed infrastructure in a single test run and automate the correlation of statistics to tenants made these experiments feasible (R1).

4.8 Service Level Agreements

Cloud providers and tenants typically negotiate SLAs to specify guaranteed levels of performance, availability, and other attributes of the service. The fulfillment or violation of service levels can be validated for each test run by the CloudPerf reporter through result checks as described in 3.3.5. Test runs are then automatically marked as successful or failed based on the configured criteria (R13).

4.9 Scalability and Overhead

Testing all aspects of cloud demands highest scalability from a performance test framework. We designed CloudPerf such that its architecture scales to largest deployments by providing horizontal scalability of load generation, parallel and asynchronous message-based three-level communication between master, prime agents, and agents, node-local storage of statistics during runtime with parallel post-processing, and many other features. In our daily work, we have challenged the implementation of CloudPerf to live up to its architectural possibilities and resolved bottlenecks where we found them. At the time of writing, we have run experiments with up to 252 tenants, 128 hosts, on servers with 1 to 384 cores per host, 72 hours of runtime, up to half a million statistic series per host with hundred millions of samples, and hundreds of GB of result data per run (R1).

The overhead of the framework can be categorized into CPU, memory, network, and disk I/O consumption. The CPU cost of load generation primarily depends on the workload's business logic, while the overhead for scheduling and timing is typically below 5% for high-level workloads (micro-benchmarks with minimal business logic may have a higher overhead). CPU cost for statistics collection depends on the invoked commands or queried APIs. The framework itself only consumes minimal CPU cycles (typically less than 0.1% of a system). The memory footprint of prime agents depends on the number of sampled statistics and usually ranges from tens to hundreds of MB. Heartbeats and statistics polling only consume negligible network bandwidth. Flushing of statistics data is batched and only generates few disk writes.

5. RELATED WORK

Numerous performance test tools and frameworks have been developed over the years, each with a different purpose in mind. Some tools such as vdbench[6], iperf[20], or uperf[5] specialize in a particular micro-benchmark and only run this single workload. YCSB[9] and YCSB++[15] are targeting cloud, but also just execute a single workload. Others tools are broader and capable of running different workloads but limited to just one domain. For example, OLTPBench[10] and SwingBench[12] only run database workloads, Cloudstone[19] only runs web workloads. JMeter[1] has expanded beyond web workloads, but is limited to supported protocols. Faban[2] is merely a harness without built-in load generation and instead leaves driver implementation to the workload. None of them support multi-tenancy. MuTeBench[13] adds multi-tenant support to OLTPBench but only supports a static SUT. CloudBench[18] emphasizes on VM deployment and automation for performance comparison of infrastructure cloud providers, but does not provide workload modeling abstractions. CloudSim[8] is a simulation framework for cloud infrastructures without tenant load generation capabilities on its own. Rally[3] is designed to benchmark OpenStack only and neither supports other cloud management software nor tenant load generation.

Many of the above tools only support steady load execution of a single tenant and require a second instance of the tool be be started to simulate either a load increase of the first tenant or activity of a second tenant. This quickly becomes impractical when frequent load changes or hundreds of tenants shall be simulated. Most tools just support closed system load generation models in which a number of threads is created to execute requests either back-to-back or with a configurable think-time. Such a model only allows to configure the concurrency of load generation, but is incapable of maintaining a configured request rate. Partly open system models implemented in some tools use cycle-time to achieve a mostly deterministic request rate, but require a sometimes impractically large number of threads if the cycle-time is high, which dramatically limits the scalability of that model. In reality however, the request arrival rate is externally given for most request-driven applications, which can only be adequately simulated with an open system model.

With the exception of those tools that focus on VM deployments, all of the above tools require a static SUT that does not change throughout test execution. They also do not support destructive stress tests in which nodes crash or reboot.

6. CONCLUSIONS AND FUTURE WORK

We have presented CloudPerf, an extensible and versatile performance test framework. Its application ranges from synthetic micro-benchmarks to high-level web or database workloads. It can be used to run performance tests against stand-alone scale-up systems as well as distributed scale-out environments. With its workload modeling abstractions it provides a high level of reuse and decouples workload development from load generation. CloudPerf supports both open as well as closed system load generation models to adequately simulate any kind of workload such as request-driven OLTP workloads, analytical or batch workloads. Its load generation capabilities range from single-tenant steady-state to dynamically changing load steps, curves or spikes for hundreds of individually controlled tenants running identical or heterogeneous workloads. CloudPerf supports elastic infrastructures and changes in the SUT such as hosts leaving or joining a test, which allows testers to conduct stress tests in which hosts are panicked or rebooted or new hosts or VMs are dynamically provisioned during a run. The load generation features of CloudPerf are complemented by an extensible statistics framework, customizable reporting, and live-monitoring of tests to allow users to analyze and correlate performance-relevant statistics across all layers of the stack. CloudPerf can be used for more traditional test scenarios where the performance of an application, database, server, or storage is evaluated or optimized, but it especially plays out its strengths in distributed and dynamic multi-tenant environments — such as *cloud*.

CloudPerf's framework has proven very versatile for all our work so far. While we keep enhancing and improving it, our future efforts mostly focus on areas beyond the framework such as the development of new workloads and services and the application of data analysis techniques on the data we capture. Especially the integration with various cloud life cycle management software through CloudPerf services will be necessary to better test and optimize cloud management. Data analytics could be used at runtime to develop and validate models for predictive forecasting of SLA violations to automatically take corrective actions, or on test results stored in a central repository to assist software and hardware architecture and design decisions for next-generation products.

7. REFERENCES

[1] Apache jmeter. http://jmeter.apache.org/. Accessed: 2016-09-17.

[2] Faban. http://faban.org. Accessed: 2016-09-17.

[3] Openstack rally. https://wiki.openstack.org/wiki/Rally. Accessed: 2016-09-17.

[4] Performance data analyzer. http://pda.nmichael.de/. Accessed: 2016-09-17.

[5] uperf. http://www.uperf.org/. Accessed: 2016-09-17.

[6] vdbench. http://vdbench.sourceforge.net. Accessed: 2016-09-17.

[7] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 9. ACM, 2009.

[8] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[10] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4), 2013.

[11] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 173–188. Springer, 2012.

[12] D. Giles. Swingbench 2.2 reference and user guide.

[13] A. Göbel. Mutebench: Turning OLTP-Bench into a multi-tenancy database benchmark framework. In *CLOUD COMPUTING 2014, The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 84–87, 2014.

[14] N. Michael and Y. Shen. Downtime-free live migration in a multitenant database. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 130–155. Springer, 2014.

[15] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 9. ACM, 2011.

[16] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.

[17] Y. Shen and N. Michael. Oracle Multitenant on SuperCluster T5-8: Scalability study. Oracle White Paper, 2014.

[18] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. Da Silva. Cloudbench: experiment automation for cloud environments. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 302–311. IEEE, 2013.

[19] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

[20] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. 2005.