The Journal of Systems and Software 116 (2016) 146–161

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

ROAR: A QoS-oriented modeling framework for automated cloud resource allocation and optimization

Yu Sun[a,*], Jules White[b], Sean Eade[c], Douglas C. Schmidt[b]

[a] Department of Computer Science, California State Polytechnic University, Pomona, USA
[b] Department of Electrical Engineering and Computer Science, Vanderbilt University, USA
[c] Siemens Industry, Germany

ARTICLE INFO

Article history:
Received 6 October 2014
Revised 20 June 2015
Accepted 6 August 2015
Available online 24 August 2015

Keywords:
Cloud computing
Resource optimization
Load testing and benchmarking

ABSTRACT

Cloud computing offers a fast, easy and cost-effective way to configure and allocate computing resources for web applications, such as consoles for smart grid applications, medical records systems, and security management platforms. Although a diverse collection of cloud resources (e.g., servers) is available, choosing the most optimized and cost-effective set of cloud resources for a given web application and set of quality of service (QoS) goals is not a straightforward task. Optimizing cloud resource allocation is a critical task for offering web applications using a software as a service model in the cloud, where minimizing operational cost while ensuring QoS goals are met is critical to meeting customer demands and maximizing profit. Manual load testing with different sets of cloud resources, followed by comparison of test results to QoS goals is tedious and inaccurate due to the limitations of the load testing tools, challenges characterizing resource utilization, significant manual test orchestration effort, and challenges identifying resource bottlenecks.

This paper introduces our work using a modeling framework – ROAR (Resource Optimization, Allocation and Recommendation System) to simplify, optimize, and automate cloud resource allocation decisions to meet QoS goals for web applications, including complex multi-tier application distributed in different server groups. ROAR uses a domain-specific language to describe the configuration of the web application, the APIs to benchmark and the expected QoS requirements (e.g., throughput and latency), and the resource optimization engine uses model-based analysis and code generation to automatically deploy and load test the application in multiple resource configurations in order to derive a cost-optimal resource configuration that meets the QoS goals.

1. Introduction

Cloud computing shifts computing from local dedicated resources to distributed, virtual, elastic, and multi-tenant resources. This paradigm provides end-users with on-demand access to computing, storage, and software services (OGRAPH and MORGENS, 2008). A number of cloud computing providers, such as Amazon Web Services (AWS) (Amazon Web Services (ELB), 2015) and Google Compute Engine (GCE) (Google Compute Engine, 2015), offer cloud computing platforms that provide custom applications with high availability and scalability. Users can allocate, execute, and terminate the instances (i.e., cloud servers) as needed, and pay for the cost of time and storage that active instances use based on a utility cost model of Rappa (2004).

To satisfy the computing resource needs of a wide variety of application types, cloud providers offer a menu of server types with different configurations of CPU capacity, memory, network capacity, disk I/O performance, and disk storage size. Table 1 shows a subset of the server configurations provided by AWS as of September 2014. For example, the *m3.medium* server with 3 ECU (i.e., EC2 Compute Unit – this is the relative measure of the CPU processing power of an Amazon EC2 cloud server) and 3.75GB memory costs $0.07/h, while the more powerful *m3.2xlarge* server costs $0.56/h. By comparison, the GCE cloud provider offers competitive options, as shown in Table 2. Although the server types are named differently (e.g., *n1-standard-1* stands for the standard general-purpose server type with 1 virtual CPU using the n1-series of machines), the resource configurations at each pricing range vary only slightly. Cloud computing users must use this information to determine the appropriate subset of these resource configurations that will run an application cost-effectively, yet still meet its QoS goals, such as response time.

* Corresponding author. Tel.: +16154579153.
E-mail addresses: yusun@cpp.edu (Y. Sun), jules@dre.vanderbilt.edu (J. White), sean.eade@siemens.com (S. Eade), schmidt@dre.vanderbilt.edu (D.C. Schmidt).

http://dx.doi.org/10.1016/j.jss.2015.08.006
0164-1212/© 2015 Elsevier Inc. All rights reserved.

Table 1
A subset of AWS EC2 instance types and pricing (Amazon EC2 Pricing, 2015).

Type	ECU	Memory (GB)	Price
m3.medium	3	3.75	$0.07/h
m3.large	6.5	7.5	$0.14/h
m3.xlarge	13	15	$0.28/h
m3.2xlarge	26	30	$0.56/h
c3.4xlarge	55	30	$0.84/h
c3.8xlarge	108	60	$1.68/h

Table 2
A subset of GCE machine types and pricing (Google Compute Engine, 2015).

Type	Virtual cores	Memory (GB)	Price (dollar/h)
n1-standard-1	1	3.75	0.07
n1-standard-2	2	7.5	0.14
n1-standard-4	4	15	0.28
n1-standard-8	8	30	0.56
n1-standard-16	16	60	1.12

A common use case of the cloud is to offer existing software products, particularly web-based applications, through a *software as a service* (SaaS) model. In an SaaS model, the SaaS provider runs the web application in the cloud and customers remotely access the software platform, while the provider manages and maintains the software. SaaS providers typically provide service level agreements (SLAs) (Shu and Meina, 2010) to their clients, which dictate the number of users they will support, the availability of the service, the response time, and other parameters. For example, a provider of a SaaS electronic medical records system may offer an SLA that ensures a certain number of employees in a hospital can simultaneously access the system and that it will provide response times under 1 s.

An important consideration of SaaS providers is minimizing their operational costs while ensuring that the QoS requirements specified in their SLAs are met. For example, in the medical records system outlined above, the SaaS provider would like to minimize the cloud resources allocated to it to reduce operational cost, while ensuring that the chosen cloud resources can support the number of simultaneous clients and response times agreed to in the client SLAs. Moreover, as new clients are added and QoS requirements become more stringent (particularly in terms of the number of supported clients), the SaaS provider would like to know how adding resources on-demand will affect the application's performance. Blindly allocating resources to the application to meet increasing load is not cost effective. Auto-scaling should instead be guided by a firm understanding of how resource allocations impact QoS goals.

Open Problem. While conventional cloud providers support simple and relatively quick resource allocation for applications, it is not an easy or straightforward task to decide an optimized and cost-effective resource configuration to run a specific application based on its QoS requirements. For instance, if a custom web application is expected to support 1000 simultaneous users with a throughput of 1000 *requests/min*, it is hard to decide the type and minimum number of servers and cloud providers needed by simply looking at the hardware configurations. Although this type of challenge existed in the traditional computing environments such as self-hosted data centers and server rooms, it has been augmented in the context of cloud computing due to the increasing number of cloud providers to choose and the virtualization-based nature of cloud computing.

Cloud providers and users have traditionally performed complex experimentation and load testing with applications on a wide variety of resource configurations. The most common practice is to deploy the application and perform a load stress test on each type of resource configuration, followed by analysis of the test results and selection of a resource configuration (Menascé, 2002). A number of load testing tools (e.g., jMeter, Apache JMeter, 2015; Halili, 2008, ApacheBench, Apache HTTP Server Benchmarking Tool, 2015, and HP LoadRunner, HP LoadRunner, 2015) are available to trigger a large amount of test requests automatically and collect the performance data. The situation gets even harder when an application includes a complex distributed architecture with multiple tiers, which requires the allocation of different groups of resources for the multiple tiers, followed by linking and connecting them properly.

Despite the importance of selecting a cost optimized resource allocation to meet QoS goals, many organizations do not have the time, resources, or experience to derive and perform a myriad of load testing experiments on a wide variety of resource types. Instead, developers typically employ a trial and error approach where they guess at the appropriate resource allocation, load test the application, and then accept it if the performance is at or above QoS goals. Optimization is usually only performed months or years into the application's life in the cloud, when insight into the affect of resource allocations on QoS goals is better understood. Even then, the optimization is often not systematic.

The primary challenges that impede early resource allocation optimization stem from limitations with load testing tools and a number of manual procedures required in the cloud resource optimization process. It is often hard to specify the customized load tests and correlate load test configuration with the expected QoS goals. Moreover, it is tedious and error-prone to manually perform load testing with different cloud resources to derive an optimized resource configuration, particularly when the application has a complex multi-tier architecture in a heterogeneous cloud platform.

Even when an optimized resource configuration is finally obtained, allocating and deploying all of the resources also requires significant orchestration and complexity. Prior research has addressed some challenges separately (e.g., modeling realistic user test behavior to produce customized load test, Draheim et al., 2006, monitoring target test server performance metrics for capacity planning,Custom Plugins for Apache JMeter, 2015). However, a comprehensive, fully automated approach designed specifically for benchmarking, deriving, and implementing optimized cloud resource allocations has not been developed, particularly for multi-tier applications.

Solution approach ⇒ *QoS-oriented modeling framework for Resource Optimization, Allocation and Recommendation System (ROAR).* To address these challenges, this paper presents a model-based system called "ROAR" that raises the level of abstraction when performing load testing and automates cloud resource optimization and allocation to transparently convert users' application-specific QoS goals to a set of optimized resources running in the cloud. A textual domain-specific language (DSL) called the *Generic Resource Optimization for Web applications Language* (GROWL) is defined to specify the high-level and customizable load testing plan and QoS requirements without low-level configuration details, such as the number of threads to use, the concurrent clients, and the duration of keeping opened connections.

The model built from the GROWL DSL can generate a test specification that is compatible with our extended version of the jMeter load testing tool (Apache JMeter, 2015; Halili, 2008). Likewise, ROAR automates the process of deploying the application to the given cloud platform, executing the load test, collecting and aligning performance metrics, analyzing the test performance model, and controlling the next test iteration. ROAR can derive the appropriate cloud resource configurations to test, as well as automatically orchestrate the tests against each resource allocation before using the results to recommend a cost-optimized resource allocation to meet the QoS goals. When the optimal configuration is decided, ROAR can also generate the resource templates to automate the final deployment process.

The work presented in this paper extends the initial prototype of ROAR (Sun et al., 2014), with the following new contributions and extensions:

- Multi-tier distributed web applications that require deployment in different groups of cloud servers can be supported by ROAR. Multi-tier configuration can be specified in the GROWL DSL. Likewise, the ROAR controller can automatically allocate the right resources to deploy and link different tiers.
- The average latency of the deployment target has been added as another key QoS goal, in addition to the target throughput supported in our earlier work. The ROAR resource optimization engine aligns the average latency metrics together with the throughput and resource utilization metrics, and filters out resource configurations that support the target throughput but fail to meet an average latency goal.
- Rather than focusing on a single cloud platform (such as AWS), the ROAR deployment manager has been extended to support deploying multi-tier applications to multiple cloud providers, which broadens its scope. In particular, we have included GCE as another platform supported by ROAR, which offers a wider range of optimization choices.
- The motivating example has been changed from a simple single-tier web application to a canonical three-tier web application with a database. Additional experiments and evaluation have also been included to showcase and quantify the performance, cost, and benchmarking effort for the resource configurations generated by ROAR.

The remainder of this paper is organized as follows: Section 2 summarizes a concrete motivating example in the context of a multi-tier web application, followed by describing the key challenges of cloud resource allocation and optimization in Section 3. Section 4 analyzes our solution by explaining each key component in the ROAR framework. Section 5 discusses the validation of this framework by presenting the generated resources for the motivating example. Section 6 compares our work on ROAR with the related research and Section 7 presents concluding remarks.

2. Motivating example

The example in this paper is based on a multi-tier web service called GhostBox built to support fine-grained control over mobile applications on the Android platform. GhostBox is designed to enable system administrators to configure specific security policies for employee smartphones and manage the policies remotely. These security policies specify the following properties:

- The types of the smartphone usage restrictions, such as running applications, accessing contacts, changing system settings and environments and
- the conditions to trigger the restrictions, which could be based on time, geographical locations, and context-aware signals (e.g., school WIFI, Bluetooth LE signals Newman, 2014).

By using GhostBox, administrators can use an administrative web portal to create and update security policies, while the smartphones used by employees periodically poll the server for updated security policies and implement the corresponding restrictions. For instance, administrators can create policies to disable gaming apps when a corporate WIFI signal is detected or enable sending/receiving text messages only from business contacts in the employers customer relationship management system.

The backend of GhostBox is built as a standard 3-tier web application. As shown in Fig. 1, the presentation tier at the top-level interacts directly with the devices and admin users. It is implemented as a Tomcat web service, and exposes HTTP APIs that can be called by the GhostBox app running in the devices that poll for device usage policies.

GhostBox also integrates a web portal as a web service to provide a simple UI for admin users to configure the policies as well as manage all the users and group settings. The middle-tier is the logical tier which handles all the data access requests sent from the presentation tier. It runs a Jetty web service (Kalin, 2013) and executes SQL

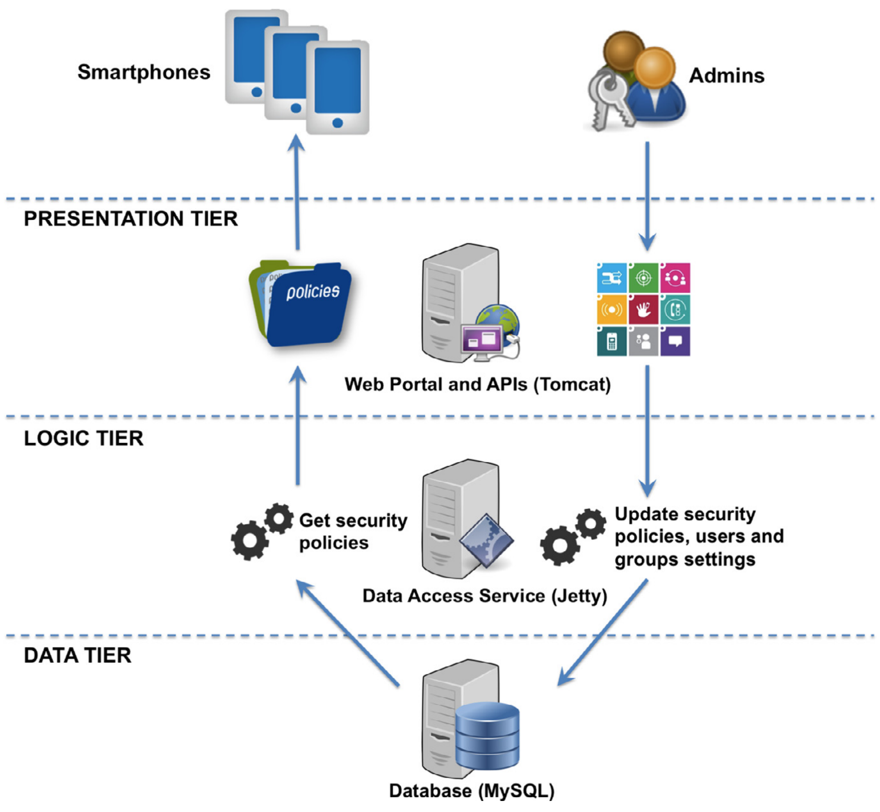
Smartphones
Admins
PRESENTATION TIER
policies
Web Portal and APIs (Tomcat)
LOGIC TIER
Get security policies
Update security policies, users and groups settings
Data Access Service (Jetty)
DATA TIER
Database (MySQL)

Fig. 1. Overview of the GhostBox multi-tier architecture.

Table 3
Key HTTP APIs provided by GhostBox backend.

Function	HTTP method	URL
Get device policies	GET	/v1/policies/{deviceId}
Get policy config page	GET	/v1/policies/{deviceId}/view
Update policies	POST	/v1/policies/{deviceId}
Get settings config page	GET	/v1/account/{accountId}/view
Update settings	POST	/v1/account/{accountId}

Table 4
The QoS eequirements of the key HTTP APIs.

Function	Expected frequency	Throughput (req/min)	Latency (ms)
Get device policies	1/5 s	24,000	100
Get policy config page	1/8 h	4.1	100
Update policies	1/8 h	4.1	100
Get settings config page	1/48 h	0.7	100
Update settings	1/48 h	0.7	100

statements to query and update the policies and other settings. A MySQL database runs in the bottom data tier to persist all the data related to policies, device usage statistics, etc.

From the perspective of the devices, the key functions provided by the GhostBox back-end are listed in Table 3. Likewise, Table 4 shows the QoS goals for each of the APIs. QoS is critical in this application since devices continuously poll the server and the service is sold based on a per-device monthly cost. The SLAs for the service specify the maximum time from when a policy is enabled by an administrator until it reaches all devices with a network connection.

Getting the device policies is the most frequently called API by all the individual devices. Each device polls for new policies every 5 s so that it can make sure updated policies are retrieved and take affect immediately. Managing the policies and accounts is done in the web portal. Admin users can login to the portal and build policy configurations. Making a change involves getting the web page resources loaded first, followed by sending an HTTP POST request to send a new policy to the server. We assume that an admin updates the policies 3 times per day, while the account settings are updated in a less frequent rate – once every 2 days.

Assume that GhostBox needs to be provisioned to support 2000 users. Based on the frequency of calls to each API from the client, it is possible to predict the minimum throughput needed to support the clients, as shown in Table 4. In addition to controlling throughput, another key QoS requirement related with the success of the application is to keep latency below a threshold. The maximum allowable latency for each function is given in the last column of Table 4.

A key requirement, therefore, is determining how to appropriately provision resources to meet the QoS requirements of the application. Determining how to provision cloud resources for GhostBox is non-trivial since it is based on a 3-tier distributed application architecture that is hard to model and characterize with existing analysis-based approaches.

3. Challenges

Determining the most cost-effective set of resources to meet a web applications QoS requirements without over-provisioning resources is a common problem in launching services in production or transitioning services to a SaaS model. The conventional practice is to benchmark web application performance using load testing on each set of possible target cloud resources. Developers use the results from these tests to estimate the required resources needed by the application to meet its QoS goals. This section summarizes the key challenges faced when deriving cost-optimized cloud resource configurations for web applications.

Challenge 1: Translating high-level QoS goals into low-level load generator configuration abstractions. To produce the huge amount of test traffic needed for QoS metrics, most tools use a multi-threaded architecture for load testing. The load test modeling abstractions from these tools are focused on the low-level threading details of the tooling rather than higher-level QoS performance specifications. For instance, jMeter, one of the most popular load testing tools, requires the specification of Thread Groups including the number of threads, the ramp-up period, and loop count.

Wrk (Wrk Modern HTTP Benchmarking Tool, 2015) takes command-line parameters to control the number of connections to keep open, the number of threads, and the duration of the test. All these OS-level details on threads create an additional level of complexity for end-users, since they must translate the desired QoS goals to test, such as a throughput target, into the right threading configurations that can generate the appropriate load profiles to determine the feasibility of these targets. Moreover, increasing the number of threads does not always increase the throughput linearly (Bacigalupo et al., 2004), which makes the translation of high-level QoS goals into low-level load testing tool configuration abstractions even harder. Developers traditionally have manually analyzed their needs to derive the appropriate threading and ramp up configurations, which is non-trivial in many cases.

Challenge 2: Resource bottleneck analysis is challenging because current tools do not collect or correlate this information with QoS metrics from tests. It is essential to understand resource utilization of allocated cloud resources in order to identify bottlenecks and make appropriate resource allocation decisions. Moreover, temporal correlation of these resource utilizations with QoS performance metrics throughout the tests is essential under the following conditions:

- When required QoS goals are not met, identifying the resource bottlenecks in order to adjust resource allocations (e.g., add more memory or switch to a more powerful CPU) is difficult.
- Even if the required QoS goals are satisfied, deriving resource utilization in order to estimate and ensure resource slack to handle load fluctuation (e.g., ensure 20% spare CPU cycles) is hard.
- When QoS goals are satisfied, ensuring that there is not substantial excess capacity (e.g., CPU utilization is at 70% or less) is challenging.
- For multi-tier web applications, the bottleneck may only be present in certain tier(s) and it is therefore important to understand the accurate resource utilization status in each individual tier and its relationship with the overall QoS or a request.

The goal is to find the exact resource configuration for each application tier where the QoS goals are met, there is a sufficient resource slack for absorbing load fluctuations, there is not too much excess capacity, and there is no more efficient resource configuration that better fits the QoS and cost goals. Manual monitoring, collection, and temporal correlation of QoS metrics with resource utilization data are tedious and often inaccurate.

Challenge 3: Increased benchmarking complexity presented in large-scale complex web applications. The resource configuration design space for an application has a huge number of permutations. A key task in optimizing resource allocations is selecting the appropriate points in the design space to sample and test in order to make resource allocation decisions. Moreover, developers must decide when enough data has been collected to stop sampling the configuration space and make a resource allocation recommendation.

To find an optimized set of cloud resources for running the application with the desired QoS, the same load test must be run multiple times against the different samples of cloud resources. By comparing the different test data and performance metrics, the final decision on the required cloud resources can be made. For a single tier web application, which can be deployed on a single host, it is straightforward to benchmark and compare the performance using different resources (e.g., different server types).

When an application involves distributing its components and tiers in different environments or hosts, however, the benchmarking complexity and effort can increase exponentially. For instance, to completely benchmark and compare the performance for the 3-tier web application in the motiving example, there will be a total of ${}_{n}C_3$ possible resource allocation configurations, where n is a the total number of available server types. If we want to analyze all of the 23 EC2 instance types provided by AWS, 1771 load tests have to be performed.

Challenge 4: Lack of end-to-end test orchestration of resource allocation, load generation, resource utilization metric collection, and QoS metric tracking. Current load testing tools only focus on the generation of loads and the tracking of a few QoS metrics. However, other key aspects of a test, such as the automated allocation of different resource configurations sampled from the resource configuration space or the collection of resource utilization data from the allocated resources, are not managed by the tools. Allocating cloud resources for complex web applications such as multi-tier web architecture requires a number of system-level configurations (e.g., security groups, load balancers, DNS names, databases) that current tools force developers to manually manage.

Although most cloud providers offer tools to automate the allocation and deployment of cloud resources, these tools require manual manipulation of many low-level details to configure. For example, Cloud Formation, (AWS Cloud Formation, 2015) provided by Amazon AWS, is an effective tool to automatically deploy an entire server stack with the cloud resources specified. However, it is a JSON-based specification that includes the configuration of over 50 resource types and hundreds of parameters, and is completely disconnected from current load testing tools. Moreover, once the resource allocations are made and tests launched, the resource utilizations on each cloud server need to be carefully remotely tracked, collected, and temporally correlated with QoS metrics tracked in the load testing tool.

Challenge 5: Extra configuration effort is required to evaluate and utilize different cloud providers. With the public cloud computing market getting increasingly popular, a number of key cloud providers offer very compelling resources and services to compete with each other. While the competition provides customers a broader range of choices on cloud resources that helps further optimize the resource configurations and pricing, extra effort is needed to carry out the benchmark and analysis across different cloud platforms, due to the different infrastructures, service features, management systems, software SDKs, etc.

For instance, accessing an AWS EC2 instance usually requires standard SSH communication with security keys, while GCE uses its own tooling and commands to login to a remote server based on OAuth2 (Jones and Hardt, 2012). Although both AWS and GCE provide a deployment template mechanism for resource allocation in JSON (the AWS Cloud Formation and Google Cloud Deployment Manager Google Cloud Deployment Manager, 2015) the format and supported type of resources vary dramatically. Handling the diversity of the cloud platforms can therefore ben an obstacle to utilize all the compelling cloud resources and make the most optimized decision.

In summary, a gap in research exists between the current load testing techniques and the resource allocation problems of cloud resource optimization. In particular, load testing aims to verify if the given resources are sufficient to meet the requirements. Conversely, the resource allocation tries to figure out exactly what are the most optimized (i.e., most cost-effective) resources to meet the requirements. In the context of multi-tier web applications and different cloud providers, this gap becomes even more substantial.

4. Solution: Resource Optimization, Allocation and Recommendation System (ROAR)

To address the five challenges presented in Section 3 associated with load testing to direct and automate resource allocation and optimization, we developed the *Resource Optimization, Allocation and Recommendation System* (ROAR). ROAR combines modeling, model analysis, test automation, code generation, and optimization techniques to simplify and optimize the derivation of cloud resource sets that will meet web application's QoS goals. This section explains each key component in the ROAR framework, which is summarized in Fig. 2.

At the heart of ROAR is a high-level textual DSL called the *Generic Resource Optimization for Web applications Language* (GROWL). ROAR uses GROWL to capture key resource configuration space and QoS goal information that is not currently captured in existing load generation tools (*Addressing Challenge 1*). The GROWL specification

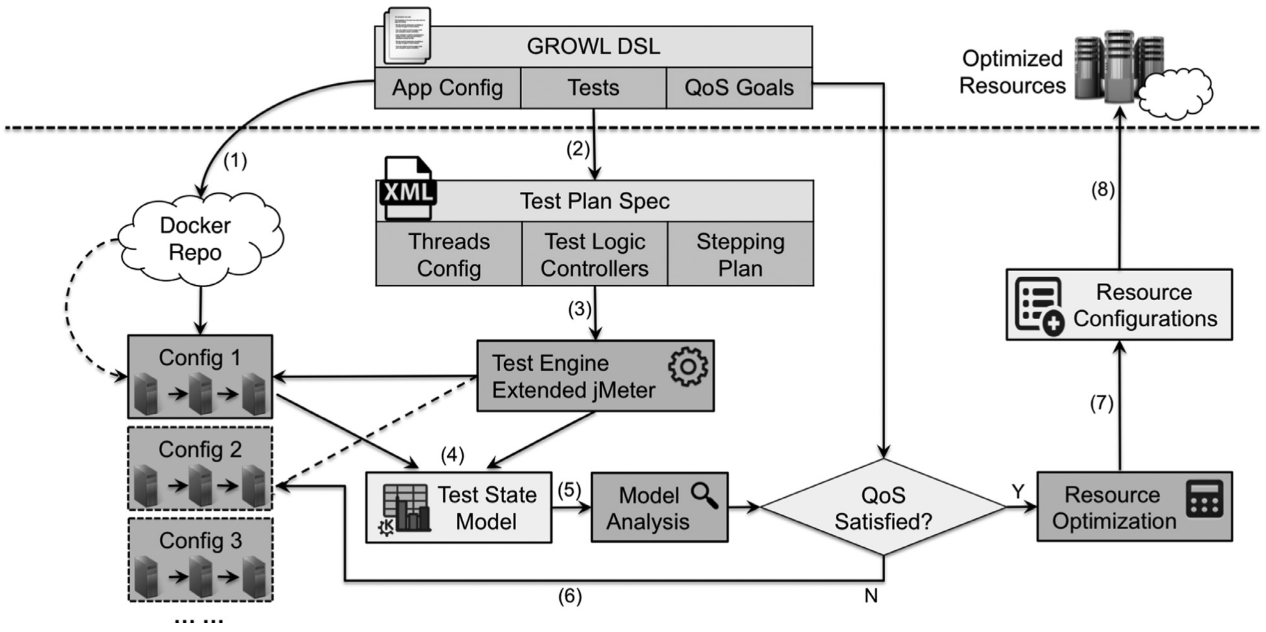

Fig. 2. Overview of the Resource Optimization, Allocation and Recommendation System (ROAR).

is translated into specific test plan inputs to the ROAR controller. ROAR then provides a fully automated end-to-end benchmarking and optimization orchestration, which handles resource allocation, application deployment, load test execution, performance metrics collection, performance model analysis, and test termination (*Addressing Challenge 4*).

ROAR can monitor detailed utilization metrics of cloud resources and align these metrics with the QoS performance metrics (*Addressing Challenge 2*). The aligned utilization/QoS metrics are then injected into a temporal performance test state model. This model can be used to perform model analysis and determine the next testing steps (*Addressing Challenge 3*).

ROAR supports conducting the same testing orchestration across different cloud platforms (e.g., AWS and GCE) based on the specification given in GROWL. When a final optimized resource configuration is determined, ROAR uses code generation to generate platform-specific deployment templates so that the allocation of the resource configurations for the chosen cloud provider can be automated (*Addressing Challenge 5*).

4.1. High-level DSL aligned with the QoS domain

Based on our experience using a variety of load testing tools, we determined the following elements of their typical abstractions that were not properly aligned with the domain of cloud resource allocation derivation:

- The low-level test thread group configurations (e.g., the number of threads, thread ramp-up time, duration of open connections).
- The complex test logic controllers based on procedural programming concepts.
- The inability to specify test application deployment and resource configurations, as well as the target QoS goals.

To address these abstraction issues, we developed the GROWL DSL to enable developers to specify web application resource configurations associated with QoS goals. GROWL will be used as the input to ROAR to automatically generate platform-specific load testing plans without requiring users to understand the underlying low-level jMeter-specific details, such as thread group or logic controllers.

Listing 1 shows the GROWL specification of the test plan for our motivating example, which contains the three major sections discussed below.

The apps section. This section describes the basic information about each of the application components (e.g., tiers) contained in the application stack to test. The attribute *platform* specifies the cloud platform to use, which could be either *aws* (i.e., Amazon Web Services) or *gce* (i.e., Google Computing Engine). A list of *app* structures describe the configurations for each application component via the following attributes:

- *id* is the identification of the application component, which is required and needs to be unique.
- *containerId* is a concept from Docker (2015) that specifies the ID of the target deployable and runnable web application container stored in Docker repository (Docker Hub Registry, 2015). To automate the application deployment to different target servers for testing, we used a containerization approach, based on the Docker container mechanism, to ease the process. Docker and the automated deployment process are discussed further in Section 4.2.
- *port* is the port number that will be used by the web application.
- *isTestEndpoint* indicates whether the application component will expose the public endpoints for the testing. Setting this attribute to *true* means that this application will be the top-tier to face users traffic directly.
- *minNumInstances* is used to ensure the availability of the application tier. If developers prefer to have a guaranteed minimum number of instances for certain tier, the number can be specified in this attribute and the resource allocation and optimization will take this into consideration.
- *env* specifies the environment variables for the operating system to be used in the allocated resources. ROAR uses this environment variable to connect different application components or tiers by letting others know how multiple applications should depend on each other. In Listing 1 an application reference (the string surrounded by the parenthesis) is used to present the actual value for the endpoint, which will be translated into the physical IP address or public DNS name in the load testing stage. The preferred multi-tier configuration and environment variable setup will be explained in Section 4.2.

The tests section. This section configures the details on what HTTP APIs or web resources to test and how to test them. The *tests* section contains a list of test samplers. Each *sampler* represents a specific type of HTTP request that we want to include in the QoS specification. The typical HTTP request configurations, such as HTTP Method (e.g., GET, POST, DELETE), HTTP URI, HTTP Headers/Parameters, HTTP Request Body can be provided for each sampler. The *percentage* attribute in each sampler controls the frequency of each sampler in the overall test plan.

For example, the percentage 99.9 means that 99.9% of all the requests in the load test will go to this API path. This attribute is the key to simplify the usage of traditional jMeter logic controllers. By default, all samplers will be executed in random order with the given percentage. However, if the *ordered* attribute in tests is set to *true*, each sampler will be executed in the order as specified in the *tests* section. The other attribute *healthCheckUrl* is not used during the load test. Instead, the ROAR test engine relies on the URL specified in this field to check if the application to test is running correctly.

The performance section. This section specifies the expected QoS goals we want to support in terms of both *throughput* and *latency*. The example in Listing 1 shows a throughput requirement for 24,050 $requests/min$ with a maximum latency to be 100 ms. GROWL has been implemented using xText (Eysholdt and Behrens, 2010). An XML-based jMeter compatible specification will be generated from a GROWL model instance as the input to the ROAR controller through xTend (Bettini, 2013).

4.2. Testing resource allocation and application deployment

The only thing developers using ROAR need to do manually is creating the GROWL model for the test plan – the rest of the testing and resource derivation process are automated for them. As shown in Fig. 2, the basic workflow of the automated end-to-end test process is to first run the web application in the most resource constrained configuration, followed by the iterative benchmarking, analysis, and optimization processes. However, deploying the multi-tier distributed web applications across the various cloud resources is not an easy task, due to the following challenges:

- Different applications require setting up different software libraries, environment configurations and even operating systems, which makes it hard to build a generic automated deployment engine.
- When trying to support deploying the same application stack to different cloud providers, the deployment mechanism needs to be adaptive to the heterogeneous cloud platforms (e.g., SDKs, authentication mechanism, configuration parameters, etc.).
- When the distributed application components depend on each other, the extra complexity is added to handle their links and correct communication.

To solve these problems, we built a container-based deployment manager as part of ROAR that can support generic application stack

```
apps {
    platform : aws;
    app {
        id : presentation-tier;
        containerId : roar-test/presentation-tier;
        port : 8080;
        isTestEndpoint : true;
        env : {
            "LOGICAL_ENDPOINT" : "{roar-test/logical-tier}"
        }
    }
    app {
        id : logical-tier;
        containerId : roar-test/logical-tier;
        port : 8900;
        minNumInstances : 2;
        env : {
            "DB_ENDPOINT" : "{roar-test/data-tier}"
        }
    }
    app {
        id : data-tier;
        containerId : roar-test/data-tier;
        port : 3306;
        isTestEndpoint : false;
    }
}

tests {
    healthCheckUrl : "/v1/ping";
    ordered : false;
    sampler {
        method : GET;
        path : "/v1/policies/testid";
        percentage : 99.9;
    }
    sampler {
        method : GET;
        path : "/v1/policies/testid/view";
        percentage : 0.01;
    }
    sampler {
        method : POST;
        path : "/v1/policies/testid";
        percentage : 0.01;
        requestBody : "{\"deviceId\":\"testid\", \"policies\":\"...\"]}";
    }
    sampler {
        method : GET;
        path : "/v1/account/testaccount/view";
        percentage : 0.01;
    }
    sampler {
        method : POST;
        path : "/v1/account/testaccount";
        percentage : 0.01;
        requestBody : "{\"accountId\":testaccount, \"settings\":\"..."]}";
    }
}

performance {
    throughput : 24050;
    timeunit : MINUTE;
    latency : 50;
}
```

Listing. 1. Test configuration example in GROWL.

deployment in various cloud platforms. The process-based container tool Docker (Fink, 2014) can package an application and its dependencies in a virtual process-based container that can run on any Linux server. Users only need to package their application in a Docker container with the preconfigured execution environment and push the application to the centralized Docker container repository.

The ROAR deployment manager will pull the application container based on the *containerId* specified in GROWL and run it automatically (Step (1) in Fig. 2). With the flexibility and portability of Docker containers, we can deploy a web application in any type of the target cloud server at anytime. The server does not need any special environment configuration except installing Docker itself, which has in fact already been included in most of the cloud VM images by the major cloud providers (AWS Elastic Beanstalk adds Docker support, 2015; Containers on Google Cloud Platform, 2014).

Another important capability provided by Docker is the ability to version containers in Git. Each change to a container can therefore be tracked, which enhances the repeatability and traceability of deployment and configuration changes. GROWL models and the generated test performance state models can also be versioned along with Docker containers to understand how the performance and resource allocations change as the application evolves.

To connect the distributed application components for all the tiers, knowing the endpoint of each component is indispensable. In

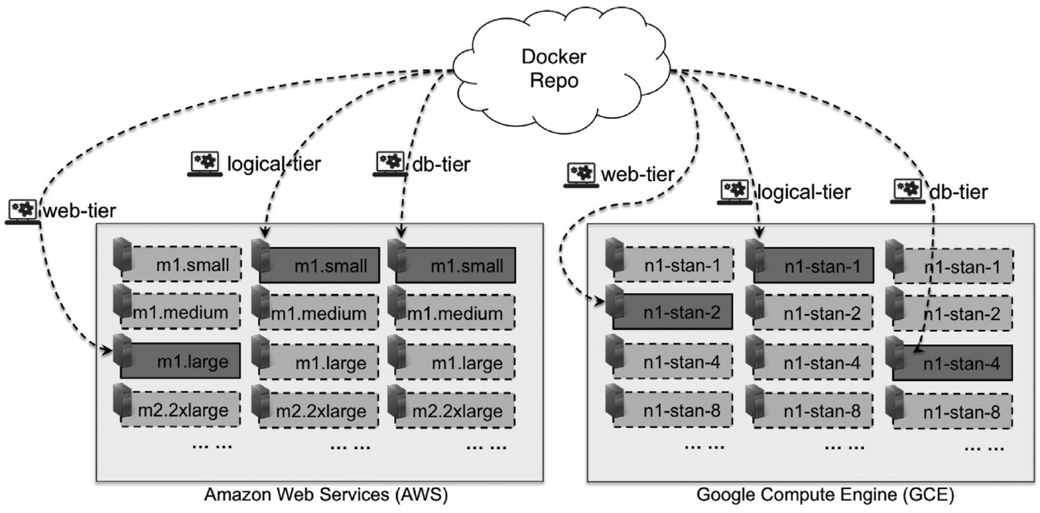

Fig. 3. ROAR's deployment manager pulls applications from Docker and deploys them to servers in different cloud platforms.

ROAR, we follow the common practice in distributed systems to configure the endpoint through environment variables, with the main benefit being that it is easy to change the value without affecting the built software artifact, and it is simple to be integrated with the deployment mechanisms (e.g., deployment scripts). As shown in Listing 1, a set of environment variables can be configured as key/value pairs in GROWL. The actual physical endpoint is not known until the cloud server has been allocated and is running. The application ID can therefore be used as a reference so that the deployment manager knows their dependency relationship, and the reference will be replaced with the final address when the cloud server is running.

The cloud resource allocation and management is done through the SDKs or APIs provided by the cloud providers. Currently, ROAR supports both AWS and GCE. When you specify the *platform* in GROWL, the ROAR deployment manager switches to use the correspondent implementation to communicate with the target platform to request the cloud resources. It first waits for the specific cloud server type to startup. When it is ready, the Docker command is triggered to pull the application container, followed by initializing the environment variables in the container with the actual IP addresses or DNS names associated with the instances. Different tiers will automatically discover each other based on the endpoint of the environment variables. Fig. 3 shows the capability of the ROAR deployment manager in this context.

4.3. Load test execution

After the entire application stack has been deployed, the ROAR test engine periodically checks the running status of the application with the *healthCheckUrl* before triggering the load test. In ROAR, jMeter is chosen to be the core test engine due to its support on a wide range of web applications, as well as its extensibility.

jMeter supports an XML-based specification for the load test plan. A GROWL interpreter has been developed to generate the jMeter specification from the GROWL model (Step (2) in Fig. 2). One part of the generated jMeter test specification, based on the *tests* section in GROWL, is the test plan, which uses a series of jMeter logic controllers to construct a test that loads the various web application paths according to the percentages or ordering specified in GROWL.

For example, the two UPDATE APIs are not called very often, so the generated jMeter test plan uses an *OnceOnlyController* to send only one request for the entire test period. The GET APIs are put inside a *ThroughputController* provided by jMeter to accurately control the execution frequency of their samplers. Finally, all four controllers are nested inside an *InterleaveController* that indicates jMeter to alternate between each of the controllers in each loop iteration. Clearly, there is a significant amount of logic and associated abstractions that are specific to jMeter and disconnected from the domain of specifying desired web application QoS.

The standard jMeter test plan only allows developers to specify the number of concurrent threads to use and provides no guarantee that the test will sufficiently load the web application to assess if the QoS goal is met. If the number of threads configured is not sufficient, jMeter will not be able to test the full capacity of the target server. However, if the number configured is too large and they ramp up too quickly, the server may run out of CPU or memory before exposing the actual throughput and server resource utilization under load. Developers must therefore rely on trial and error specification of these parameters to test each target throughput.

To overcome this issue, we developed a model analysis to determine the appropriate test strategy to assess the target throughput goal. To produce test loads with the desired throughput accurately, we built a customized jMeter plugin (Custom Plugins for Apache JMeter, 2015) and throughput shaper that analyzes and operates on the GROWL model. The shaper is capable of dynamically controlling the number of threads being used to ensure that the test meets the GROWL goals. Based on the needed throughput, it automatically increases or decreases the number of threads to reach the throughput accurately, while simultaneously watching the resource utilization of the target server to ensure that it isn't overloaded too quickly.

Our throughput shaper supports gradual throughput increments. A throughput stepping plan is automatically derived from the specified throughput QoS goals in GROWL. For instance, if the target throughput goal is 5000 *requests*/*min*, the generated test plan divides the test into 10 different stepping periods, so that the test load increases from 0 to 500 *requests*/*min* within the first 5 s and then keeps running at 500 *requests*/*min* for another 20 s before jumping to the next level at 1000 *requests*/*min*.

The test load is gradually increased, rather than maximizing the load from the start, in order to avoid crashing the server if the load is substantially more than the capacity of the server. In addition, the incremental ramping periods provide more diverse sampling metrics used in the test model analysis in the next step. The test plan specification is used by the extended jMeter engine to trigger the load test (Step (3) shown in Fig. 2).

4.4. Test performance data collection and alignment

The key function of jMeter is to collect, aggregate, and summarize the QoS performance metrics for the applications being tested, such as throughput, latency, and error rate. What is missing from jMeter, however, is the capability of monitoring and collecting resource utilization metrics from the servers used for hosting the application. A key component of ROAR is thus a mechanism to collect and correlate server resource utilization with QoS performance metrics.

The ROAR throughput shaper discretizes the test epoch into a series of discrete time units with known QoS values. The QoS values at each discrete time unit are correlated with server resource utilization during the same time unit. To record the actual throughput in each moment, we modified the native jMeter *SimpleReporter* listener to be capable of automatically reporting the throughput every single second (the native *SimpleReporter* only reports the current aggregated throughput). Likewise, to record the target server performance metrics, another customized jMeter plugin (Custom Plugins for Apache JMeter, 2015) is applied, which runs a web-based performance monitor agent in the target server. HTTP calls can be made from jMeter to retrieve the performance metrics at runtime and assign them to the appropriate time unit for correlation with QoS values.

A range of detailed resource utilization metrics is supported by our plugins. However, we only record CPU, memory, network I/O and disk I/O, since we have found these are the primary resource utilization metrics for making resource allocation decisions. The extensions we made to the jMeter plugin include (1) using a global clock that is consistent with the time used by the QoS performance records, (2) enabling the collection process in non-GUI jMeter mode, and (3) distributed resource utilization collection for each application using the global clock.

Based on the global clock, the collected resource utilization and QoS metrics are aligned and stored in a test state model (Step (4) in Fig. 2). The test state model is the input to the bottleneck analysis and resource optimization algorithms run later and can also be loaded in the performance viewer to display the results visually. As shown in Fig. 4, based on the global clock, the correlation of QoS (throughput and latency) and resource utilization (CPU/Memory/Network/Disk utilization) can be clearly seen.[1]

4.5. Model analysis and termination strategy

The test state model containing the global clock-aligned QoS values and resource utilizations allows us to analyze resource bottlenecks and derive the best cloud resource configurations to meet the modeled QoS goals. One issue that must be addressed before resource configuration derivation is potential points of instability in the data, which show an inconsistent correlation between the QoS and the resource utilization with the values recorded during the rest of the test epoch. These periods of instability are due to jMeter's creation and termination of a larger number of threads in the beginning and end of each testing step triggered by the throughput shaper.

4.6. Handling performance instability

To handle the instability, a short 10-s warmup period has been automatically added in the beginning of each test plan. The 1/10th of the target throughput load will be used during the warmup and the collected metrics during this period of time will be ignored. After the warmup, the test plan gradually increase the test loads by 1/10th of the target throughput and keeps running it for 20 s. Based on our experiments, the performance and utilization metrics usually fluctuate during the first half of each individual step which does not reflect the real QoS that can be sustained by the application at that load. Thus, the final test performance model only uses the second half of the data to average the values for both QoS performance and resource utilization metrics.

[1] Disk I/O is not involved with this application.

Therefore, if n is the number of total steps used in the test plan, after load testing, a series of aligned data set on the actual throughput (T), average latency (L), CPU utilization rate (U_{cpu}), memory utilization rate (U_{mem}), network utilization rate (U_{net}) and disk utilization rate (U_{disk}) will be available in the following test performance model:

$$\begin{aligned}
T &= \{t_i | 1 \le i \le n\} \\
L &= \{l_i | 1 \le i \le n\} \\
U_{cpu} &= \{u_{cpu_i} | 1 \le i \le n\} \\
U_{mem} &= \{u_{mem_i} | 1 \le i \le n\} \\
U_{net} &= \{u_{net_i} | 1 \le i \le n\} \\
U_{disk} &= \left\{u_{disk_i} | 1 \le i \le n\right\}
\end{aligned}$$

4.7. Resource allocation analysis

The resource allocation analysis can be performed with a variety of models and optimization goals (Lin et al., 2010; Warneke and Kao, 2011; Wei et al., 2010). ROAR uses an optimization engine to select the minimum number of servers N needed to run each application tier to reach the expected throughput within the expected latency.

Let T_{ex} be the expected throughput and L_{ex} be the expected maximum latency specified in GROWL. If T_p is the actual peak throughput with a certain resource configuration, then:

$$N = \lceil T_{ex}/T_p \rceil \tag{1}$$

The key problem here is to find a reasonable T_p from all the data points in the test state model without violating the expected latency goal T_{ex} and the resource utilization limits. Generally, when the peak throughput is reached, one or more resource utilizations will also approach 100% utilization (e.g., CPU or memory utilization goes to 100%, or network usage goes to 100%), and at the same time, the average latency of responses increases dramatically. We use these two conditions to determine T_p. Formally,

$$\begin{aligned}
&T_p = \max_{0 \le i \le n} t_i \\
\textit{where},\quad & l_i \le L_{ex} \\
& u_{cpu_i} < 100\% \\
& u_{mem_i} < 100\% \\
& u_{net_i} < 100\% \\
& u_{disk_i} < 100\%
\end{aligned}$$

Fig. 4 shows a typical dataset collected during a test. The throughput shaper produces an increasing number of requests as indicated by the expected throughput (yellow line in the top graph of Fig. 4), but the actual throughput fails to meet the QoS goal after the 5th step (blue line in the top graph of Fig. 4). Checking the performance metrics of the target server for each application tier, it can be found that although memory and network have not been fully utilized, the CPU utilization in Tier 2 (Logical Tier) has reached its limit – 100%, and therefore the peak throughput using this server appears at the 6th step.

A complete test state model provides the detailed QoS performance with the given resource configuration for each tier. In order to compare and figure out the most optimized resource configuration, the same load testing process needs to be repeated for each type of resource to get the different values for N. The optimization engine in ROAR is capable of automating all the tests sequentially from the most resource constrained server types to more powerful and more expensive types (Step (6) in Fig. 2).

In Amazon EC2, resources can only be allocated in fixed configurations of memory, CPU, and network capacity, rather than in arbitrary configurations. There are over 20 different instance types, which are virtual machines with fixed resource configurations, provided by AWS, so it is hard to manually sample the performance of all instance

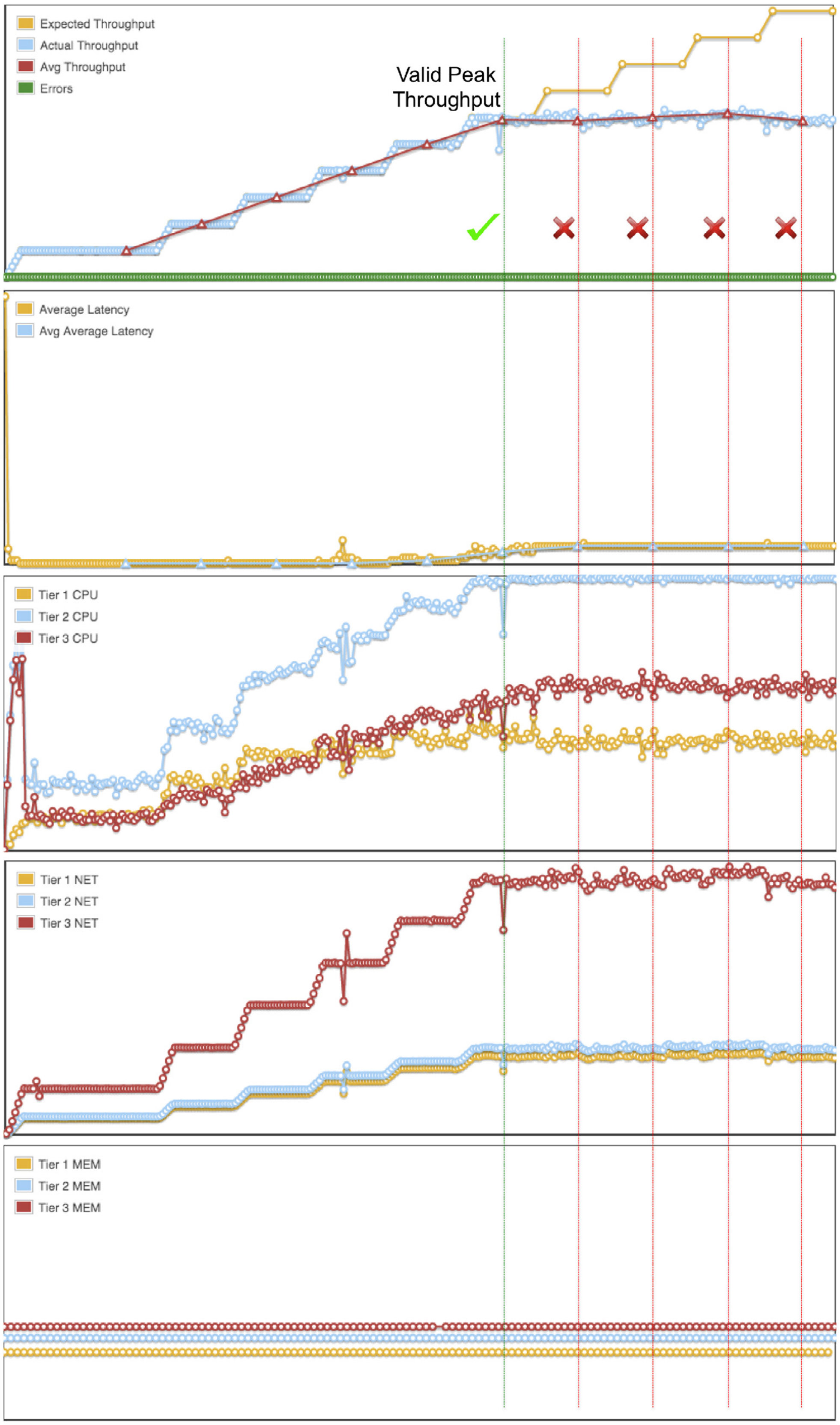

Fig. 4. A sample test state model with aligned data on throughput, average latency, CPU, memory, and network. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article).

types. In addition, when the web application stack involves multi-tier applications like the motivating example, benchmarking all the possible combination of the server types with each tier would be tedious and costly.

After the test state model is generated, the ROAR optimization controller uses the test output data to choose the next resource configuration and then switches a tier to a bigger type of server with a more powerful hardware configuration if resource bottlenecks were found in that tier. To reduce the benchmarking effort, if the target QoS goals are met, the test can terminate after the next resource configuration is tested. The reason that the test can terminate is that any larger resource configurations will only have additional slack resources that are not needed to meet the desired QoS goals.

Conversely, when the target QoS goals are not met with a certain configuration, the ROAR test engine analyzes which application tier caused the bottleneck, i.e., the application tier with at least one type of resource utilization that reached 100%. ROAR will only change the instance type of that tier for the next test iteration until the QoS goals are met or other tiers become the bottlenecks. Using this approach, ROAR can effectively permute the resource configuration of the application to efficiently find bottlenecks without exploring all possible resource configurations.

The entire testing process is repeated to derive a model of the QoS and resource utilization on the new resource configuration. This process is repeated until the most cost-effective configuration solution is found. A prototype of a web-based service has been implemented to take the GROWL as the input and generate the final cloud resource configurations ready for deployment.

The final resource configuration is decided by comparing and analyzing the results (Step (7) in Fig. 2). The most cost-effective solution is derived by finding the minimum total cost of using each type of server to reach the desired throughput. Let I be the set of applications, and J be the set of available server types. If P_{ij} represents the cost of the server type j ($j \in J$) which is chosen to run application i ($i \in I$), the total cost is $C = \sum_{i \in I, j \in J} P_{ij} * N_i$, and the most cost-effective solution is Min_C.

Besides finding the most cost-effective solution, knowing the solution that supports the fastest response time (i.e., the lowest response latency) is also essential for general web applications in practice. The fastest response time configuration is decided by the average latency measured for each type of server. Although the latency varies based on the testing environment (e.g., running the load test inside AWS would have a much lower latency than running the test outside of AWS), the comparison of the average latencies for different server types could produce a good indicator of the performance with which two different servers can handle the same request load. Thus, we choose the server type with the lowest average latency as the fastest response time configuration.

ROAR also supports the guarantee of availability for application tiers. As explained in GROWL specification, the attribute *minNumInstances* defines the minimum number of instances that should be applied to the given tier. Therefore, if the number of instances generated in the final resource configuration based on the throughput and latency requirements is less than the number of minimum instances, the ROAR engine will replace the generated configuration with the minimum number of instances, and use that to compare the cost.

4.8. Automated resource template generation

After the cloud resource allocation needed to meet the QoS goals is derived, ROAR can then automatically allocate the resources and deploy the web application. Manually allocating resources with a fleet of servers in a cloud environment is not an easy task in general. It's even harder when optimized service architecture patterns and best practices are needed.

For example, in AWS, the best practice of deploying multiple servers to support the same web application is to use an Elastic Load Balancer (ELB) (Amazon Web Services - Elastic Load Balancing, 2015) as the entry point. An ELB is capable of handling any number of requests from users and balancing the dispatch of the requests to each server behind it. A number of configuration parameters for security groups, private keys, availability zones, server launch configuration data, and DNS names have to be specified correctly when using an ELB.

To ease the resource allocation process, ROAR uses a generative approach to produce the configuration artifacts needed for each cloud provider. Currently, ROAR supports artifact generation for both AWS and GCE, and support for OpenStack is planned. The AWS artifact generator is built on top of the AWS Cloud Formation service, which provides an engine for automating the deployment of multi-server architectures via templates defined in JSON. Based on the N and server types generated from the resource optimization engine, we fill the template with the appropriate configuration steps, resource types, and other parameters needed to launch the application.

After the template is generated, it can be uploaded to the AWS Cloud Formation console, allowing developers to deploy the web application in the derived configuration with a single click (Step (8) in Fig. 2). A similar mechanism called the Cloud Deployment Manager (CDM) (Google Cloud Deployment Manager, 2015) is provided by GCE, which also uses JSON template but contains different configurations since the cloud resource types and management SDK varies dramatically. GCE also provides a load balancing service to include multiple servers and provides a single endpoint for each tier. The template generation is capable of filling the same type of cloud resource configuration in the platform-specific templates.

5. Evaluation

This section presents a case study evaluation based on the motivating example from Section 2. Based on the given web application stack, we run experiments with two sets of QoS goals in both cloud platforms, as shown in Table 5.

These experiments allow us to observe the different optimizations provided by ROAR, as well as the difference between the various cloud computing platforms.

By default, ROAR considers all the available instance types from each cloud platform. This experiment focuses on the general-purpose cloud servers that fit the nature of the motivating example. The goal is to clearly compare the differences in the same instance type category, as well as avoid the expense of the high-end server types (e.g., GPU or Cluster instance types, high memory or high I/O instance types). The instance types used from AWS include: *m3.medium, m3.large, m3.xlarge, m3.2xlarge, c3.4xlarge and c3.8xlarge*, while in GCE we choose: *n1-standard-1, n1-standard-2, n1-standard-4, n1-standard-8 and n1-standard-16*, as listed in both Tables 1 and 2. The chosen sets of servers in both platforms are very competitive in terms of both pricing and hardware configurations.

5.1. Resource optimization results

The test plan was specified using GROWL for each of the 4 tests to run, as shown in Listing 1. ROAR then automates the benchmarking

Table 5
The list of test QoS goals and platforms used to evaluate ROAR.

Test no.	Target throughput (req/min)	Target latency (ms)	Cloud platform
1	24,050	50	AWS
2	24,050	50	GCE
3	50,000	30	AWS
4	50,000	30	GCE

Table 6
Resource benchmarking result for Test 1.

Tier	Instance types	Peak throughput (req/min)	Average latency (ms)	Total server required	Total cost (dollar/h)
Data	m3.medium	24,050	11	1	0.07
Data	m3.large	24,050	10	1	0.14
Logical	m3.medium	6500	13	4	0.28
Logical	m3.large	19,000	12	2	0.28
Logical	m3.xlarge	24,050	10	2	0.56
Logical	m3.2xlarge	24,050	10	2	1.12
Presentation	m3.medium	8100	14	3	0.21
Presentation	m3.large	20,000	13	2	0.28
Presentation	m3.xlarge	24,050	13	1	0.56
Presentation	m3.2xlarge	24,050	11	1	0.56

Table 7
Resource benchmarking result for Test 2.

Tier	Instance types	Peak throughput (req/min)	Average latency (ms)	Total server required	Total cost (dollar/h)
Data	n1-standard-1	24,050	12	1	0.07
Data	n1-standard-2	24,050	10	1	0.14
Logical	n1-standard-1	12,000	17	3	0.21
Logical	n1-standard-2	17,000	16	2	0.28
Logical	n1-standard-4	24,050	14	2	0.56
Logical	n1-standard-8	24,050	13	2	1.12
Presentation	n1-standard-1	13,000	17	2	0.14
Presentation	n1-standard-2	18,500	16	2	0.28
Presentation	n1-standard-4	24,050	16	1	0.56
Presentation	n1-standard-8	24,050	14	1	0.56

Table 8
Resource benchmarking result for Test 3.

Tier	Instance types	Peak throughput (req/min)	Average latency (ms)	Total server required	Total cost (dollar/h)
Data	m3.medium	24,500	33	3	0.21
Data	m3.large	48,000	28	2	0.28
Data	m3.xlarge	50,000	22	1	0.28
Logical	m3.medium	6500	38	8	0.56
Logical	m3.large	19,000	37	3	0.42
Logical	m3.xlarge	28,000	25	2	0.48
Logical	m3.2xlarge	46,500	20	2	1.12
Logical	c3.4xlarge	50,000	19	2	1.68
Logical	c3.8xlarge	50,000	20	2	3.36
Presentation	m3.medium	8100	39	7	0.49
Presentation	m3.large	20,000	30	3	0.42
Presentation	m3.xlarge	31,000	30	2	0.48
Presentation	m3.2xlarge	50,000	29	1	0.56
Presentation	c3.4xlarge	50,000	25	1	0.84

and optimization processes, with the results shown in Tables 6, 7, 8 and 9. Each table lists the peak throughput and average latency data collected for each tier/instance type in different test iterations. Knowing the peak throughput, we can estimate the total number of servers required for each tier using Eq. (1) and then calculate the total cost. Using the minimum cost option for each tier (i.e., the gray row in each table), the most cost-effective resource allocation can be derived.

For example, when running *Test* 1 in AWS, the best resource combination is 1 *m3.medium* instance for the *data-tier*, 2 *m3.large* instance for *logical-tier* and 3 *m3.medium* instances for the *presentation-tier*. Conversely, when running *Test* 1 in GCE, the most optimized solution would be 1 *n1-standard-1* instance for the *data-tier*, 3 *n1-standard-1* instances for *data-tier* and 2 *n1-standard-1* instances for the *data-tier*. All of these critical pieces of information are derived automatically using GROWL and ROAR and would be difficult to obtain through current load testing practices with existing tools and approaches. One thing to be noted on the generated logical tier configuration is that a

Table 9
Resource benchmarking result for Test 4.

Tier	Instance types	Peak throughput (req/min)	Average latency (ms)	Total server required	Total cost (dollar/h)
Data	n1-standard-1	27,000	31	2	0.14
Data	n1-standard-2	43,000	28	2	0.28
Data	n1-standard-4	50,000	24	1	0.28
Logical	n1-standard-1	12,000	40	8	0.56
Logical	n1-standard-2	17,000	38	3	0.42
Logical	n1-standard-4	27,000	25	2	0.48
Logical	n1-standard-8	46,000	24	2	1.12
Logical	n1-standard-16	50,000	24	2	1.68
Presentation	n1-standard-1	12,500	40	7	0.49
Presentation	n1-standard-2	18,000	38	3	0.42
Presentation	n1-standard-4	29,500	31	2	0.48
Presentation	n1-standard-8	50,000	23	1	0.56
Presentation	n1-standard-16	50,000	22	1	0.84

minNumInstance of 2 has been specified to guarantee the availability, so that the ROAR engine automatically use 2 instances even if one instance might be sufficient for the throughput and latency requirements. This can be seen in *logical-tier m3.xlarge* and *m3.2xlarge* configurations in *Test*1, as well as *logical-tier n1-standard-4* and *n1-standard-8* configurations in *Test*2.

Most of the cloud providers precisely scale their server performance according to pricing. It is, therefore, likely that a configuration with multiple servers of smaller instance types can meet the same QoS goals as a configuration with fewer servers of bigger types. For instance, Table 6 shows that multiple *m3.medium* or multiple *m3.large* servers can reach the same performance as a single *m3.xlarge* for *logical-tier* with the same cost. In this case, we often consider choosing the configuration with lower request latency.

Conversely, if fewer servers are preferred (e.g., to reduce operational and management complexity), the ROAR optimization engine can be configured to prioritize solutions with fewer server instances. For example, a single server for the database could be used to avoid a complex database clustering configuration. The results for *Test*3 and *Test*4 show that when the requirements on latency are changed to 30 ms, some server types are automatically filtered out when the benchmarked latency is above this threshold.

Regarding the optimized solution, we currently rely on the direct aggregation of the throughput results to estimate the maximum throughput when multiple instances of the same type are being used. In fact, a further optimization could be done in this space by mixing different types of servers. For instance, Table 8 shows a more cost-effective solution for the *presentation-tier* uses 2 *m3.large* plus 1 *m3.medium*. However, mixing different types of servers can create challenges in practice, because the load balancers equally distribute traffic by default, which means that smaller server types may receive the same amount of traffic as bigger ones and therefore the overall performance may not meet expectations.

5.2. Performance validation

Satisfying all the QoS performance requirements is a key goal for any resource allocation task. It is essential to ensure that the generated cloud resource configuration by ROAR can actually handle the desired throughput and latency requirements. Although detailed benchmarking processes have been conducted for different configurations, the performance of aggregating all the applications and resources together is still based on estimation. To validate the final performance and QoS, therefore, performance experiments were done for all of the 4 test scenarios. Each test scenario was deployed using the generated resource configuration templates – load balancers with cloud servers behind them.

Table 10
Average performance deviation (%).

Test no.	Throughput (req/min)	e_{tp} (%)	Latency (ms)	e_l (%)
1	23,950	0.4	24	0.0
2	24,580	0.0	26	0.0
3	50,100	0.0	29	0.0
4	49,900	0.02	27	0.0

A dedicated load test was triggered for each test scenario based on the given QoS goals. By comparing the actual throughput and average latency to the ROAR estimated throughput and latency, we derived the average error rate of the throughput (e_{tp}) and latency (e_l) for each application in the experiment, as shown in Table 10. It can be seen that the estimates on throughput are within 0.4% of their actual values.

The 0.4% error for *Test*1 arises largely from the error of the throughput shaper used in the load test engine. For instance, when the provisioned throughput is 24,050 $requests/min$, the load generated may not exactly reach 24,050 $requests/min$ but instead top out at 23,950 $requests/min$ due to thread contention, which produces a 0.4% error rate. Regarding the latency error rates, they are all 0%, meaning that the average latency constraints were always properly derived by the ROAR generated resource configurations.

The low error rates were expected because the throughput and resource allocations were based on actual load testing and benchmarking, with the only estimation being aggregating the peak throughput to support the target throughput.

One thing to be noted about performance is that ROAR targets resource optimization for the web applications whose performance are bounded by CPU, Memory and I/O capacities, rather than the internal algorithm logic or other critical resources. In other words, the performance of the application has a linear relationship with the number of servers (i.e., resources) provided. This applies to most of the web application scenarios that are based on the load balancing cloud architecture. If the performance of the application is not linearly determined by CPU, memory or I/O capacities, when ROAR performs the benchmarking against different types of machines, a non-linear performance improvement will be observed, so that when generating the final resource configuration, the more expensive and bigger types of VMs will be automatically filtered out based on the current algorithm.

One useful step we can add to ROAR framework is an automatic validation process after the resource configuration has been generated. We can automatically trigger the expected load and verify if the generated configuration can satisfy the QoS goals, so that developers can be notified with the issue if the generated resources do not meet the requirements.

5.3. Benchmarking effort

As described in Section 4.5, ROAR automatically scales up the type of instance used by an application tier when that tier reaches its utilization limit for one or more resources and terminates the test when the QoS goals are met. As given in Section 3, the total number of configuration combinations needed to test the motivating example would be 444 for *Test*1 and *Test*3 using 6 AWS instances, 300 for *Test*2 and *Test*4 using 5 GCE instances. By comparison, Table 11 shows the number of total benchmarking iterations required for each test.

These results indicate that ROAR effectively reduced the benchmarking effort by using bottleneck analysis to navigate the resource allocation configuration space of each tier.

5.4. Cost of resource configurations

Knowing the most optimized resource allocation in each cloud provider for different QoS goals can provide users with critical guidance on cloud provider selection. Table 12 shows that the cost of using AWS and GCE is only slightly different based on the QoS goals.

Table 11
Benchmarking effort comparison.

Test no.	Number of iterations to reach QoS goals	Maximum number of potential iterations
1	10	444
2	10	300
3	14	444
4	13	300

Table 12
Cost comparison.

Test no.	Cloud platform	Total cost (Dollars/H)
1	AWS	0.56
2	GCE	0.42
3	AWS	1.18
4	GCE	1.32

This slight difference is expected because both cloud providers offer the same categories of servers at the same price ranges. Although their marketed hardware specification and parameters are different, their basic computing capabilities are very similar.

However, we did notice that a performance difference existed between the two cloud providers. Particularly, for the motivating example web application used in this paper, the *n1-standard-1* instance from GCE always performed much better than the server in the same pricing range *m3.medium* from AWS. However, for the higher end servers in AWS, such as *m3.xlarge* and *m3.2xlarge*, they always showed better performance in terms of both throughput and latency than the equivalent GCE instance types.

6. Related work

This section compares our work on ROAR with the following related research efforts.

Ferry et al. (2013) summarized the state-of-the-art on cloud optimization and pointed out the need for model-driven engineering techniques and methods to aid provisioning, deployment, monitoring, and adaptation of multi-cloud systems. Revel8or (Zhu et al., 2007) is a model-driven capacity planning toolsuite to solve the problems related to complex multi-tier applications with strict performance requirements. They use UML 2.0 to model and annotate design diagrams, and derive performance analysis models. However, Revel8or only supports applications developed in a model-driven approach from platform-independent model to platform-specific model to the final code, while our approach targets all web applications, including hand-written and legacy applications. TOSCA is another DSL proposed by Binz et al. (2014) to specify the topology and orchestration configuration for cloud applications to ease the cloud management tasks. Unlike our work, however, it does not focus on resource optimization and allocation.

A number of research projects have focused on model-based load testing. Draheim et al. (2006) used a modeling approach to produce realistic load testing scripts for web applications. Their approach simplifies the creation of realistic usage models of individual user behavior based on stochastic models. However, they do not analyze the minimum required cloud resources. Wang et al. (2013) present a *Load Testing Automation Framework* (LTAF), which offers usage models to simulate users's behavior in load testing and workflow models to generate the realistic testing load. Similar to prior work, LTAF focused on simplified creation of realistic load testing configurations for web applications, with detailed performance reports. Since resource optimization is not their goal, neither of these two frameworks takes the

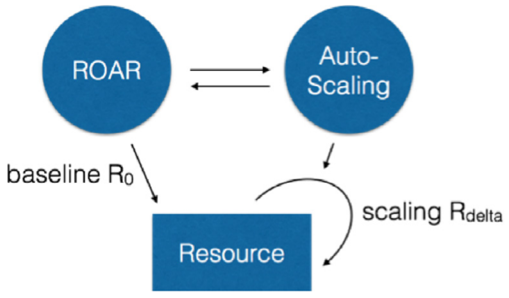

Fig. 5. ROAR assists auto-scaling in cloud resource allocation and optimization.

target server resource consumption into consideration, in contrast to our work on ROAR.

Wei et al. (2010) applied game theory to solve cloud-based resource allocation problems using abstract resource and application mathematics models. Based on a specification of the application's QoS constraints, Binary Integer Mining and evolutionary algorithms are used to drive the optimization. This research differs from our work on ROAR in the following ways:

- it focuses on resource allocation for multiple applications being scheduled in large cloud environments and
- it emphasizes abstract analysis rather than empirical testing of production applications.

Li et al. (2009) present a method for resource optimization in clouds by using performance models in the deployment and operations of the applications running in the cloud. An optimization algorithm is implemented to accommodate different goals, scopes and timescales of optimization actions, to minimize cost while meeting SLAs across a large variety of workloads. Recent work by Chaisiri et al. (2012) focused on the cloud resource optimization problem from a different perspective. They examined provisioning algorithms to better utilize market-based resources that fluctuate in cost based on demand to reduce total cost. To handle the resource optimization for multiple applications, Frey et al. (2013) presented a search-based genetic algorithm to find the near-optimal solution that optimizes response times, costs, and number of SLA violations. Their approach is based on the simulation tool called CDOSim to improve the search process, while our approach focuses on performing actual benchmarking to obtain the accurate data. Instead of using genetic algorithms, Catan et al. (2013) specified application dependencies and criteria, as well as applied external constraint solvers to reach an optimal configuration. Although the framework they built can also handle multiple-tier web application, they focus on finding out the most optimized actions to transform the deployment configuration from one state to another when a reconfiguration request comes in, rather than targeting the QoS goals.

Instead of focusing on the high-level QoS goals such as throughput and latency, Yi et al. (2014) concentrated on the low-level network infrastructure for cloud computing environments and introduced a multilayer optical network architecture that could monitor the resource status and guarantee the network bandwidth at runtime. Another interesting work done by Di and Wang (2013a) is the dynamic resource optimization for self-organized clouds. This type of cloud computing is similar to the idea of volunteer computing, where each individual could contribute their machine to the collaborative cloud environment with the P2P networking techniques. The major problem they solved is the task resource allocation under a certain budget, rather than the performance. Mao et al. (2010) and Mao and Humphrey (2011) did a comprehensive analysis on using auto-scaling technique to ensure the optimized resource allocation with the budget constraints. They modeled the computing jobs, workflows, and deadlines, which provides an effective guidance for the auto-scaling algorithms. These two papers target a different type of problem from ours – given a list of jobs to finish with deadline, how to arrange them with the auto-scaled resources. Similarly, the work done by Di and Wang (2013b) also tries to perform the optimization using the deadline-based models, which helps to ensure the completion of tasks under deadline with the minimum cost.

Another important related area to compare is the auto-scaling mechanisms provided by most of the cloud providers, such as AWS Auto-Scaling (Amazon Web Services - Auto-Scaling, 2015) and GCE Autoscaler (Google Compute Engine – Autoscaler, 2015). For instance,

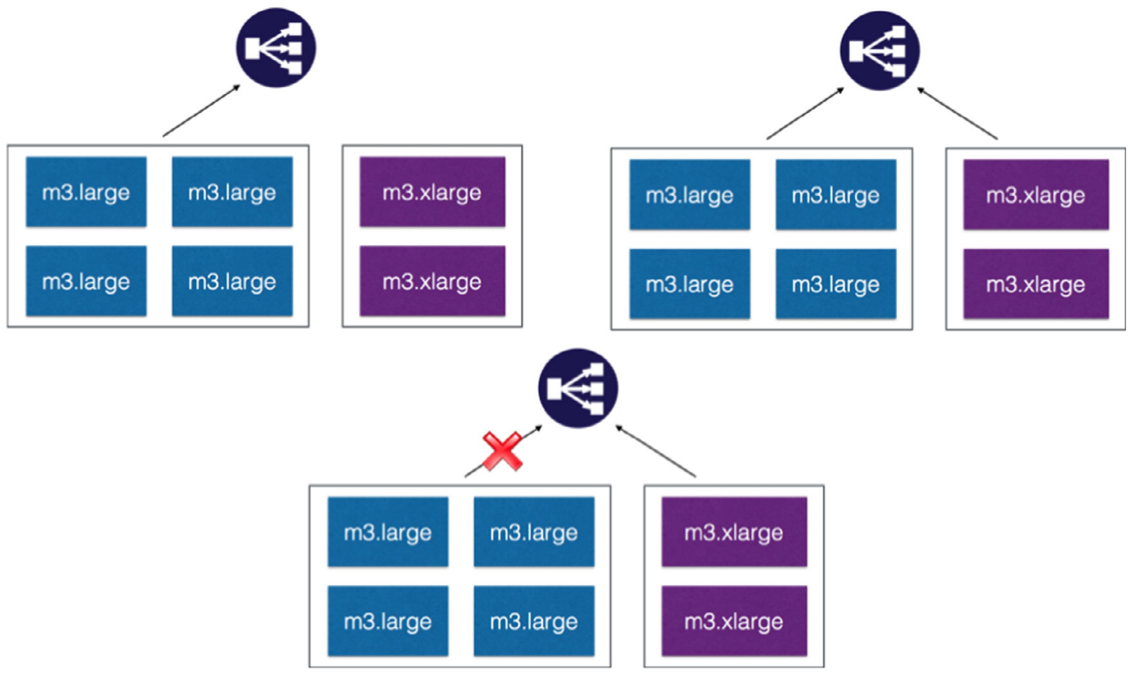

Fig. 6. An overview of rearranging the resource configuration: (1) creating the new set of cloud resources; (2) connecting the new set of cloud resources to the load balancer and (3) disconnecting the old set of cloud resources from the load balancer.

AWS Auto-Scaling allows developers to configure auto-scaling policies to automatically scale the capacity up or down according to load conditions defined. Autoscaler performs auto-scaling based on average CPU usage, customized cloud metrics, and the load balancer capacity. However, while auto-scaling can be applied to adapt to the load changes automatically and dynamically, it has some limitations and does not solve all the practical issues related with application capacity planning, resource allocation and optimization. Specifically, auto-scaling cannot precisely determine and optimize the baseline resource for the initial capacity allocation, which may affect the optimum of deploying, maintaining and auto-scaling applications. On the other hand, the auto-scaling policies based on adjusting the number of instances are designed to adapt to the changes occurred in the monitored metrics, without considering the overall optimum of the resource configuration in terms of cost.

ROAR is designed to be an enhancement and complement to the existing cloud deployment and auto-scaling mechanism as shown in Fig. 5, because it can be applied to help determine the optimal initial baseline resource configuration to ensure the safe but also the optimized application deployment. Once the application runs, ROAR assists auto-scaling engine to figure out the optimized auto-scaling policies with more options. As a next step to assist auto-scaling, we have made an API to query the most optimized resource configured based on the input QoS requirements, so that the auto-scaling engine can call the API to get the most optimized configuration at runtime. When a rearrangement of the resource is needed, it can be done efficiently by launching the new resources, adding the new resources to the load balance, and terminating the old service resources as shown in Fig. 6.

7. Concluding remarks

This paper presents a QoS-oriented modeling framework called the *Resource Optimization, Allocation and Recommendation System* (ROAR). ROAR automates the testing and derivation of optimized cloud resource allocations for web applications. In particular, ROAR automates the end-to-end test orchestration from application deployment, metrics collection and alignment, to test model generation and the test iteration with termination.

ROAR uses a textual DSL called GROWL to hide the low-level configuration and analysis details of load testing for the complex multi-tier web applications. QoS goals (including target throughput and latency) can be expressed via the DSL, which will be used to direct the generated load test plans. GROWL generates an optimized resource configuration to meet the web application's QoS goals, as well as a deployment and configuration template for the specific cloud provider.

Our future work focuses on enriching GROWL and exposing more best practices in load testing and resource optimization as simple DSL language constructs. Examples include supporting complex test behaviors that are driven by input parameters (e.g., generate random or variable parameters as test inputs) or time-based tests (e.g., send request r1 during the period of t1, and request r2 only during the period of t2).

Moreover, only average latency is currently supported as one of the QoS goals. Therefore, enabling other goals (such as percentile latency) would also be useful. While multiple-tier application can be automatically benchmarked and optimized, we assume that each tier is deployed separately in different servers. In some cases, co-locating the different application tiers together in the same host or migrating applications across different cloud platforms might offer a deeper optimization space. However, more sophisticated optimization algorithms would be needed to handle this case.

Finally, extending ROAR to support auto-scaling at runtime and performance validation is another key research direction for the future. Currently, ROAR can be connected to the server monitors in the cloud, through the optimization query API so that whenever a new set of QoS requirements is needed, the ROAR optimization engine can be triggered at runtime to re-configure the allocated resources (e.g., adding/removing resources). In fact, using ROAR to configure the auto-scaling policies has a big advantage over the traditional auto-scaling mechanism provided by most of the cloud platforms – the auto-scaling policies can be based on containers with finer granularity (particularly with the co-locating different applications in the same server), rather than at the VM level which could potentially over-provision the resources.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2015.08.006 .

References

Amazon EC2 Pricing, 2015. http://aws.amazon.com/ec2/pricing/.
Amazon Web Services - Auto-Scaling, 2015. http://aws.amazon.com/autoscaling/.
Amazon Web Services, 2015. https://aws.amazon.com/.
Amazon Web Services - Elastic Load Balancing, 2015. https://aws.amazon.com/elasticloadbalancing/.
Apache HTTP Server Benchmarking Tool, 2015. http://httpd.apache.org/docs/2.2/programs/ab.html.
Apache JMeter, 2015. http://jmeter.apache.org/.
AWS Cloud Formation, 2015. http://aws.amazon.com/cloudformation/.
AWS Elastic Beanstalk adds Docker support, 2015. http://aws.amazon.com/about-aws/whats-new/2014/04/23/aws-elastic-beanstalk-adds-docker-support/.
Bacigalupo, D.A., Jarvis, S.A., He, L., Nudd, G.R., 2004. An investigation into the application of different performance prediction techniques to e-commerce applications. In: Proceedings of the IEEE 18th International Parallel and Distributed Processing Symposium. IEEE, p. 248.
Bettini, L., 2013. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd.
Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., 2014. Tosca: portable automated deployment and management of cloud applications. In: Advanced Web Services. Springer, pp. 527–549.
Catan, M., Di Cosmo, R., Eiche, A., Lascu, T.A., Lienhardt, M., Mauro, J., Treinen, R., Zacchiroli, S., Zavattaro, G., Zwolakowski, J., 2013. Aeolus: mastering the complexity of cloud application deployment. In: Service-Oriented and Cloud Computing. Springer, pp. 1–3.
Chaisiri, S., Lee, B.-S., Niyato, D., 2012. Optimization of resource provisioning cost in cloud computing. IEEE Trans. Serv. Comput. 5 (2), 164–177.
Containers on Google Cloud Platform, 2015. https://cloud.google.com/compute/docs/containers.
Custom Plugins for Apache JMeter, 2015. http://jmeter-plugins.org/.
Di, S., Wang, C.-L., 2013. Dynamic optimization of multiattribute resource allocation in self-organizing clouds. IEEE Trans. Parallel Distrib. Syst. 24 (3), 464–478.
Di, S., Wang, C.-L., 2013. Error-tolerant resource allocation and payment minimization for cloud system. IEEE Trans. Parallel Distrib. Syst. 24 (6), 1097–1106.
Docker, 2015. https://www.docker.io.
Docker Hub Registry, 2015. https://registry.hub.docker.com/.
Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., Weber, G., 2006. Realistic load testing of web applications. In: Proceedings of the IEEE 10th European Conference on Software Maintenance and Reengineering, CSMR. IEEE.11–pp
Eysholdt, M., Behrens, H., 2010. Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. ACM, pp. 307–309.
Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A., 2013. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: Proceedings of the IEEE 6th International Conference on Cloud Computing, pp. 887–894.
Fink, J., 2014. Docker: a software as a service, operating system-level virtualization framework. Code4Lib J. 25.
Frey, S., Fittkau, F., Hasselbring, W., 2013. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: Proceedings of IEEE International Conference on Software Engineering. IEEE Press, pp. 512–521.
Google Cloud Deployment Manager, 2015. https://cloud.google.com/deployment-manager/.
Google Compute Engine – Autoscaler, 2015. https://cloud.google.com/compute/docs/autoscaler/.
Google Compute Engine, 2015. https://cloud.google.com/products/compute-engine/.
Halili, E.H., 2008. Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites. Packt Publishing Ltd.
HP LoadRunner, 2015. http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/.
Jones, M., Hardt, D., 2012. The OAuth 2.0 Authorization Framework: Bearer Token Usage. Technical Report. RFC 6750, October.
Kalin, M., 2013. Java Web Services: Up and Running. O'Reilly Media, Inc..

Li, J., Chinneck, J., Woodside, M., Litoiu, M., Iszlai, G., 2009. Performance model driven QoS guarantees and optimization in clouds. In: Proceedings of IEEE Workshop on Software Engineering Challenges of Cloud Computing, ICSE. IEEE Computer Society, pp. 15–22.
Lin, W.-Y., Lin, G.-Y., Wei, H.-Y., 2010. Dynamic auction mechanism for cloud resource allocation. In: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid). IEEE, pp. 591–592.
Mao, M., Humphrey, M., 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, p. 49.
Mao, M., Li, J., Humphrey, M., 2010. Cloud auto-scaling with deadline and budget constraints. In: Proceedings of the 11th IEEE/ACM International Conference on Grid Computing (GRID). IEEE, pp. 41–48.
Menascé, D., 2002. Load testing of web sites. IEEE Internet Comput. 6 (4), 70–74.
Newman, N., 2014. Apple ibeacon technology briefing. J. Direct Data Digit. Mark Pract. 15 (3), 222–225.
OGRAPH, B.T., MORGENS, Y.R., 2008. Cloud computing. Commun. ACM 51 (7).
Rappa, M.A., 2004. The utility business model and the future of computing services. IBM Syst. J. 43 (1), 32–42.
Shu, Z., Meina, S., 2010. An architecture design of life cycle based SLA management. In: Proceedings of the 12th International Conference on Advanced Communication Technology (ICACT), 2. IEEE, pp. 1351–1355.
Sun, Y., White, J., Eade, S., 2014. A model-based system to automate cloud resource allocation and optimization. In: Model-Driven Engineering Languages and Systems. Springer, pp. 18–34.
Wang, X., Zhou, B., Li, W., 2013. Model-based load testing of web applications. J. Chin. Inst. Eng. 36 (1), 74–86.
Warneke, D., Kao, O., 2011. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. IEEE Trans. Parallel Distrib. Syst. 22 (6), 985–997.
Wei, G., Vasilakos, A.V., Zheng, Y., Xiong, N., 2010. A game-theoretic method of fair resource allocation for cloud computing services. J. Supercomput. 54 (2), 252–269.
Wrk Modern HTTP Benchmarking Tool, 2015. https://github.com/wg/wrk.
Yi, P., Ding, H., Ramamurthy, B., 2014. Cost-optimized joint resource allocation in grids/clouds with multilayer optical network architecture. IEEE/OSA J. Opt. Commun. Netw. 6 (10), 911–924.
Zhu, L., Liu, Y., Bui, N.B., Gorton, I., 2007. Revel8or: model driven capacity planning tool suite. In: Proceedings of the 29th IEEE International Conference on Software Engineering, ICSE. IEEE, pp. 797–800.

Dr. Yu Sun is an Assistant Professor of Computer Science in the Computer Science Department at California State Polytechnic University, Pomona. His research focuses on cloud computing, mobile computing, Internet-Of-Things (IoT) and large-scale distributed systems. Before joining Cal Poly Pomona, he was a post-doc research associate at Vanderbilt University, the Director of Engineering at Cloudpoint Labs, where he led the research and development on the high-precision 3D augmented reality technology for mobile platforms. He also worked in Amazon Web Services as a software development engineer and participated in the development of the first cloud-based mobile web browser project Amazon Silk. His research is conducted through the Software engineering, Cloud and Mobile computing (SoftCoM) Laboratory at Cal Poly Pomona, which he directs. He received his Ph.D. from the University of Alabama at Birmingham in 2011.

Dr. Jules White is an Assistant Professor of Computer Science in the Department of Electrical Engineering and Computer Science at Vanderbilt University. He was previously a faculty member in Electrical and Computer Engineering at Virginia Tech and won the Outstanding New Assistant Professor Award at Virginia Tech. His research has won 3 Best Paper Awards and 2 Best Student Paper Awards. He has also published over 95 papers. Dr. White's research focuses on securing, optimizing, and leveraging data from mobile cyber-physical systems. His mobile cyber-physical systems research spans four key focus areas: (1) mobile security and data collection, (2) high-precision mobile augmented reality, (3) mobile device and supporting cloud infrastructure power and configuration optimization, and (4) applications of mobile cyber-physical systems in multi-disciplinary domains, including energy-optimized cloud computing, smart grid systems, healthcare/manufacturing security, next-generation construction technologies, and citizen science. His research has been licensed and transitioned to industry, where it won an Innovation Award at CES 2013, attended by over 150,000 people, was a finalist for the Technical Achievement at Award at SXSW Interactive, and was a top 3 for mobile in the Accelerator Awards at SXSW 2013. His research is conducted through the Mobile Application computinG, optimizatoN, and secUrity Methods (MAGNUM) Group at Vanderbilt University, which he directs.

Dr. Douglas C. Schmidt is a Professor of Computer Science at Vanderbilt University. His research interests include patterns, optimization techniques, and empirical analyses of middleware and domain-specific modeling tools that facilitate the development of distributed real-time and embedded systems. He received his Ph.D. in 1994 from the University of California, Irvine.