UPSARA: A Model-driven Approach for Performance Analysis of Cloud-hosted Applications

Yogesh D. Barve*, Shashank Shekhar†‡, Shweta Khare*, Anirban Bhattacharjee* and Aniruddha Gokhale*
*Dept of EECS, Vanderbilt University, Nashville, TN 37212, USA
Email:{yogesh.d.barve,shweta.p.khare,anirban.bhattacharjee,a.gokhale}@vanderbilt.edu
†Siemens Corporate Technology, Princeton, NJ, 08540, USA
Email:shashankshekhar@siemens.com

***Abstract*—Accurately analyzing the sources of performance anomalies in cloud-based applications is a hard problem due both to the multi tenant nature of cloud deployment and changing application workloads. To that end many different resource instrumentation and application performance modeling frameworks have been developed in recent years to help in the effective deployment and resource management decisions. Yet, the significant differences among these frameworks in terms of their APIs, their ability to instrument resources at different levels of granularity, and making sense of the collected information make it extremely hard to effectively use these frameworks. Not addressing these complexities can result in operators providing incompatible and incorrect configurations leading to inaccurate diagnosis of performance issues and hence incorrect resource management. To address these challenges, we present UPSARA, a model-driven generative framework that provides an extensible, lightweight and scalable performance monitoring, analysis and testing framework for cloud-hosted applications. UPSARA helps alleviate the accidental complexities in configuring the right resource monitoring and performance testing strategies for the underlying instrumentation frameworks used. We evaluate the effectiveness of UPSARA in the context of representative use cases highlighting its features and benefits.**

***Index Terms*—Performance analysis, model-driven, DSML, Interference, Cloud, Resource Management, Performance Monitoring, Benchmarks.**

I. Introduction

Multi-tenancy in cloud deployment and changing workload patterns for cloud-based applications make it hard to analyze and diagnose any incurred performance issues and find appropriate solutions to resolve them. In particular, identifying the sources of performance interference due to multi-tenancy remains a hard problem [1], [2], [3]. To support our hypothesis, consider Figure 1, which shows the performance variabilities incurred by a multi-tenant cloud deployment of an image recognition application based on Inception-Resnet v2 Keras machine learning model [4]. As seen, the performance deteriorates significantly when running in a multi-tenant environment compared to a baseline performance with no resource contention. Understanding these performance issues, which themself can change dynamically, is important to make effective dynamic deployment and resource management decisions so as to meet applications' service level objectives (SLOs).

‡Work done as a part of doctoral studies at Vanderbilt University

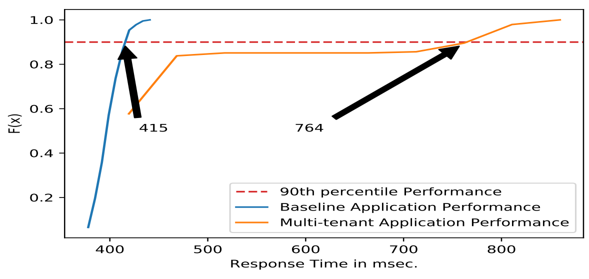

Fig. 1. CDF of Response Time for Inception-ResNet v2 model using Keras with 4 Cores

A number of resource- and application performance-monitoring frameworks have been developed in recent years, e.g., Nagios [5], Zabbix [6], Intel SNAP, linux-perf, collectd [7] and systat to name a few. These performance monitoring tools provide the users with different system-level and in some cases application-level metrics, which in turn provide insights into the runtime performance of these applications.

As is the case with any technology, using such tools often involves a steep learning curve in understanding their usage (e.g., their APIs), their features and capabilities, and often requires users to manually write custom configuration scripts and programs to effectively utilize the tools. With advances in hardware that enable more finer-grained performance metrics to be collected, these problems are further excarbated. For example, Intel has recently introduced the Cache Monitoring Technology tools which are compatible with the new generation of Intel architectures [8]. To use these capabilities, new monitoring tools and programs need to be added to capture the desired performance statistics. Further, a user must possess expert knowledge about the hardware architecture and monitoring tools.

Recent efforts [9], [10], [11], [12] have attempted to address these concerns. However, there still remain many unresolved issues that must be addressed. For instance, such solutions are usually tightly coupled to an execution platform and as such do not support compatibility with newer platforms. Secondly, many tools are non-intuitive to use and less user friendly thereby requiring users to manually configure and install the monitoring probes on the runtime platforms, which is an error-prone process. Beyond addressing these limitations, newer capabilities are needed to enable runtime performance monitoring of these applications. For instance, older frameworks often cannot be extended to exploit finer-grained or newer hardware metrics, such as last-level cache utilization or non uniform memory access (NUMA) patterns.

To overcome the above challenges, we propose to utilize the principles of software product lines (SPLs) [13] in creating a model-driven generative framework called UPSARA-*Understanding **P**erformance of **S**oftware **A**pplications and **R**untime System **A**nalysis.* A product line is essentially a family of product variants which have a set of features common to all variants, but differ from each other along some other features of an overall feature set that defines a product line. Our approach is based on the observation that different monitoring frameworks share many common features, while at the same time differ on some other key features. Thus, the different monitoring frameworks can be seen as variants of a product line. Using generative capabilities provided by model driven engineering (MDE) [14], we synthesize the configurations for the different frameworks and automate the desired performance monitoring tasks for the use case under consideration.

The rest of the paper is organized as follows: Section II presents the motivation, requirements and the architecture of UPSARA. Section III delves into the details of the model driven engineering techniques, and the UPSARA domain specific modeling language. Section IV describes the generative capabilities of UPSARA. We present validation of UPSARA using representative usecases in Section VI. Related work is described in Section VII. Finally, we present concluding remarks and future directions for UPSARA in Section VIII.

II. Design and Implementation of UPSARA

In this section we highlight the key challenges and solution requirements, followed by an overview of the UPSARA solution.

A. Eliciting Challenges and Solution Needs

The stakeholders of UPSARA are cloud performance engineers who would like to analyze an application's performance on the target platform. An application can be a monolithic application such as a database, a micro-service based component assembly, or can be a distributed application such as Map-Reduce, distributed co-simulations, or parallel processing jobs. To analyze the performance delivered to an application, an engineer needs to collect system metrics such as the CPU, network, disk, memory utilization as well as micro-architectural metrics, such as context switches, cache utilization, memory interconnect utilization, among others. These metrics are needed to pinpoint performance interference issues incurred due to multi tenancy. Moreover, the engineer will also need application-level performance metrics, such as observed response times and throughput.

The system metrics collection is typically performed using external programs such as collectd, statsD, likwid, linux-perf, Intel PMU tools, Intel RDT tools, among others [15]. Such tools can measure some or all of the metrics of interest. As such, one may need a single or a collection of such tools to cover the spectrum of metrics of interest. The runtime platform and the application then needs to be configured accordingly with the right collection of tools based on the metrics selected by the performance engineers. Each tool may impose different approaches for its configuration such as through *.conf* files, or by passing input parameters during program invocation. The target platform also might have limitations as to which metrics it can offer to be monitored, and what tools need to be executed to capture the supported metrics.

Based on these concerns, below we describe the key requirements for a solution like UPSARA that we present in this paper.

1) ***Ease of Use***: The metric instrumentation framework should have a lower entry barrier so that it is easy to use in the continuous integration and performance studies of a cloud-hosted application. The steep learning curve for individual tools hinders the users from making use of the features provided by such platforms. Thus, UPSARA should provide intuitive and higher-level abstractions, which hide the lower-level complexity thereby making it more easy for the end users to utilize the platform.
2) ***Ensuring the correctness of the configuration:*** It is important that the metrics selection on a platform are supported by that platform. For instance, monitoring *non-uniform memory access* (NUMA)-level statistics or measuring cache statistics requires that the underlying platform hardware support these capabilities. Without such support, forcing the monitoring tool to capture these parameters will either result in capturing garbage data or throw a run-time exception complaining about the non-existence of such features. Hence, UPSARA should natively support built-in correctness or a violation checker, which will enforce *correct by construction* design.
3) ***Well-formedness of generated artifacts:*** To avoid writing low-level code artifacts, generative programming has helped developers by synthesizing various code artifacts based on user-defined templates. Although, generative programming can synthesize these artifacts, it is essential that a generative solution adopted by UPSARA be correct and adhere to the domain-specific rules. This is necessary to ensure functional correctness of the system.
4) ***Support for heterogenous runtime architectures***: UPSARA should be able to support heterogeneous runtime architectures, which are common in cloud environments. Since each such runtime architectures might have their own set of metric monitoring tools, UPSARA should support seamless and automated composition of such tools in its architecture.
5) ***Extensibility and tool reuse:*** UPSARA must be able to reuse the existing capabilities of underlying measurement tools that provide monitoring of system metrics rather than reinventing the wheel by developing new monitoring

tools. The framework should support semantics for easy plug and play architecture for adding support for new tools. Also, as new hardware architectures get developed, UPSARA should be able to add support for new architecture metric measurements.

B. Architecture and Workflow

Figure 2 illustrates the high-level operational workflow of UPSARA that performance engineers can use for analyzing their cloud-hosted applications. To begin with, the designer of the experiment provides a high-level specification of the performance analysis to be conducted. The specification includes the various metrics to be measured, the type of application to be studied, and information about the runtime platform on which the application needs to be executed. The designer encodes this specification using the visual elements of UPSARA's domain-specific modeling language (DSML) (See Section III). Once the specification is captured, UPSARA transforms the specification into valid configuration scripts and deploys the artifacts onto the runtime platform. Application execution then begins on the target runtime platform. The system metrics and the application metrics are captured during the execution of the application and are made available to the user by means of auto-analysis and visualization charts.

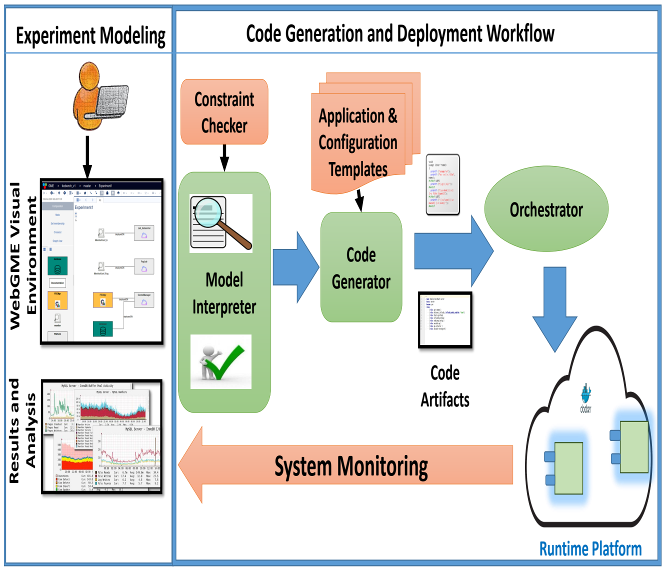

Fig. 2. High Level Overview of UPSARA

We now describe the main building blocks of UPSARA.

- *WebGME Visual Environment:* This block provides an intuitive interface via the DSML for designing and orchestrating application performance analysis experiments. The DSML-based interface enables (1) configuring various system metrics to be collected on the target runtime platform, (2) controlling the execution of the experiments, and (3) providing analysis and real-time system dynamics.
- *Model Interpreter:* The model interpreter validates the well-formedness of the user-supplied model by traversing the model elements and validating their syntactic and semantic correctness while also generating the desired artifacts. The generative aspects of the interpreter handled by the Code Generator block (described below) implements a graph traversal logic using a visitor design pattern [16] and synthesizes artifacts that capture the relationships and model attributes as described by the user in lower-level representation.
- *Constraint Checker:* This module is responsible for detecting any violation in the design of the experiment. It consists of rules which specify the correct properties of the application execution. Examples of constraints are *an application can be deployed only on one runtime platform at a time* or *supported list of metrics of the runtime platform.* Thus, if the designer of the experiment constructs a scenario where the application is deployed on two different platforms at once, a constraint violation action will be triggered notifying the user of the violations. These constraints are captured and are made available in the rules database of UPSARA.
- *Code Generator:* This component is responsible for generating the desired artifacts as specified by the user-supplied model. The code generator has access to a repository of application and configuration template schemas as required by the underlying monitoring and deployment tools. As an example, the generator can synthesize the schemas for the metric monitoring software. Also, based on the application deployment on the target runtime system, the code generator can generate scripts called *playbooks* that will then be input to an application orchestration framework called *Ansible* [17]. The code generator is also responsible for generating visualization artifacts for viewing the results.
- *Orchestrator:* This component interfaces with the runtime platform. The orchestrator is responsible for deploying the generated artifacts on the target platform by instantiating appropriate application deployment and metrics collection.
- *Result and Analysis:* This component is responsible for presenting the user with the analysis and statistics of the application performance as designed by the user. It features Python notebooks hosted on a Jupyter server [18]. The graphical visualization allows users to analyze various application performance characteristics. These include the resource consumption for different system metrics such as CPU, load, memory, energy, etc. Specialized analyses modules can be integrated into the generated notebooks which provide relevant analysis for the selected frameworks.

III. UPSARA's Domain-Specific Modeling Language (DSML) Design

We now delve into the details of UPSARA's domain-specific modeling language (DSML).

A. Encoding UPSARA's Product-line Feature Model

We leverage model-driven engineering (MDE) techniques to encode the different elements involved in UPSARA's product line for aiding in performance analysis of cloud applications. Specifically, the feature model is encoded as a meta-model(s) of an underlying DSML. To design the DSML, we have used the WebGME modeling environment [19] to create the meta-models, writing model interpreters, and encoding generative logic for synthesizing artifacts. WebGME itself is a cloud-based service that provides a browser-based design environment and supports creating visual DSMLs.[1]

UPSARA's visual DSML provides several benefits when compared to manually written scripts. The visual component blocks provide an easy way to instrument and construct an experiment which can then be deployed as specified [20], [21], [22]. This reduces barrier to entry for developers that can rapidly and concisely describe and start running experimental scenarios without learning a new programming language terminology and syntax [23]. The visual blocks are intuitive and hence the learning curve for using UPSARA is negligible. Next, we describe the main aspects of the UPSARA DSML and their responsibilities.

1) UPSARA Main Meta-model: Figure 3 shows the main building block of the DSML. It includes the following meta-model components:

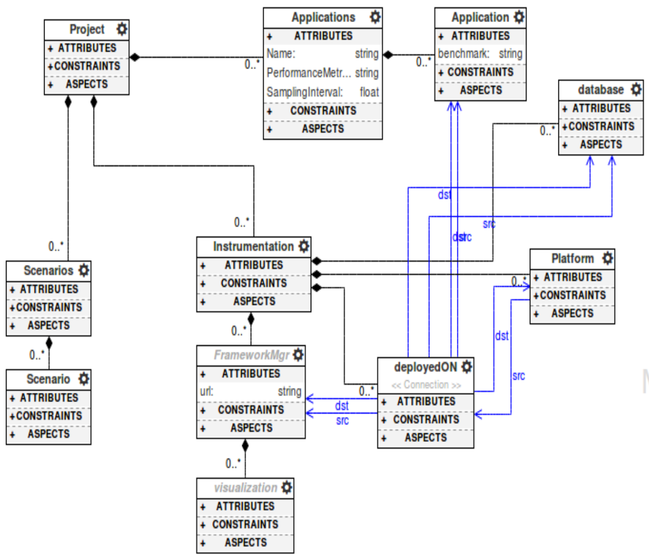

Fig. 3. UPSARA Main Meta-model

Project: represents the top-level meta-model block of UPSARA. *Project* comprises an *application* set, and includes information about the *instrumentation* metrics to be collected on the runtime *platform*. The *Project* component also contains a *scenarios* model. *Scenarios* describe different deployment schemes for running applications on the runtime platform.

[1]A textual DSML is an alternative approach but we did not explore it.

Platform: The platform represents the cloud platform on which the application performance study will be conducted. This usually represents target host machines which includes: virtual machine, lightweight containers (e.g., Docker), bare-metal, as well as a cluster of host machines.

FrameworkMgr: This component represents the monitoring framework which the user can configure with the desired set of metrics to monitor. Individual frameworks can be custom-built with a predefined set of metrics that are supported for the specific performance monitoring activity. Specialized frameworks can be built by extending the *FrameworkMgr* to suit different application performance monitoring scenarios.

Instrumentation: This component defines how the metrics collection will be configured on the underlying cloud platform using one or more concrete frameworks, e.g., collectd. The metrics are associated with *frameworkmgr*. Moreover, the runtime platform is represented by *platform*. The relationship between *frameworkmgr* and the *platform* is represented by the connection semantics in WebGME.

deployedON: This represents connection semantics in WebGME which capture the relation between two nodes. Here *deployedON* represents a connection between *FrameworkMgr* and *Platform*, *database* element and *platform*.

Scenarios: As mentioned earlier, this component includes the different deployment options for running the applications on cloud platforms. A user may want to study application performance on multiple runtime platforms at once. For such usecases, the user will define a scenario that captures the *application to the runtime platform* mapping. *Scenario* component facilitates describing such mapping for the performance study.

Visualization: Visualization of data and results is provided as an important aid in conducting performance analysis of applications. The *visualization* component generates plots that are specific to the *FrameworkMgr* being developed. Every *FrameworkMgr* instance can have a custom set of visualization artifacts that meaningfully represent the performance analysis data for the framework. Visualization can be produced using a set of Jupyter notebooks [18] that capture specifics of the application performance.

Database: This component is used for storing the collected performance metrics data.

2) UPSARA Metric Meta-model: Figure 4 showcases a snippet of the metric monitoring meta-model. The system metric to be monitored can be macro-level metrics such as CPU, memory, network, disk, load, as well as micro-architectural metrics such as the context switches, L1 cache, L2 cache, L3 or LLC cache utilization, cache hit and miss counts, scheduler specific metrics, Docker container-specific metrics, etc. Figure 4 shows a snippet of the micro-architectural meta-model. As shown in Figure 4, using attribute selection we allow fine-grained access to specific aspects of the system metrics that a user would want to collect. As an example, the figure shows that cache-miss property can be set to *TRUE* or *FALSE*.

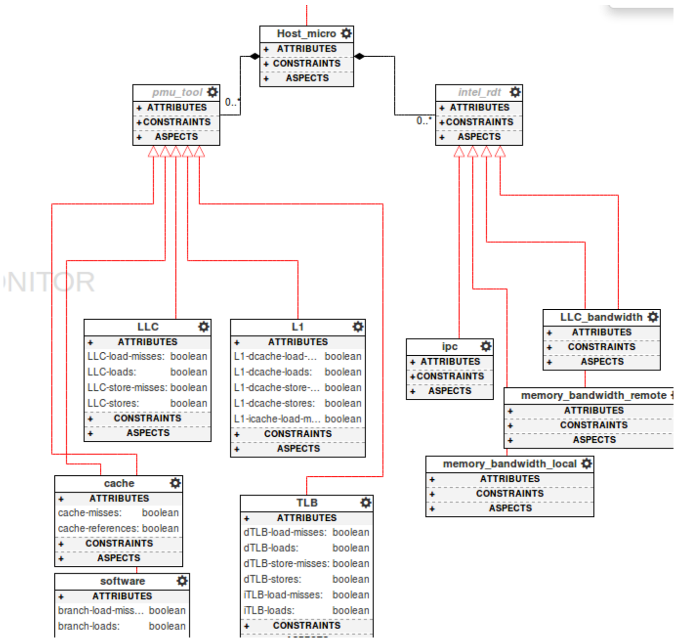

Fig. 4. Snippet of UPSARA Micro-metric Meta-model

B. *UPSARA Platform Meta-model*

Figure 5 illustrates the cloud platform meta-model. *Platform* represents the target runtime platform on which the application performance needs to be studied. *Platform* could be any of baremetal host machine (*Host*), a virtual machine (*VM*), or a lightweight Docker container (*Docker*). All these are derived from an abstract *machine* concept in the DSML. *Cluster* comprises a group of *machine* elements. A *cluster* could be either in the form of a *Datacenter*, *FogPlatform*, or an *EdgePlatform* that represents the clouds, fog and edge computing resources, respectively [24].

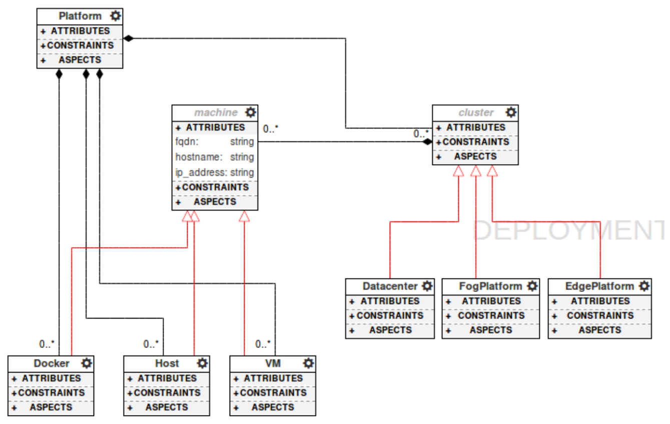

Fig. 5. Snippet of UPSARA Runtime Platform Meta-model

1) UPSARA Application Deployment Scenario Meta-model: To test the performance of an application on various platforms we provide an application deployment *scenario* meta-model. The scenario meta-model shown in Figure 6 illustrates how an *application* maps to runtime platforms.

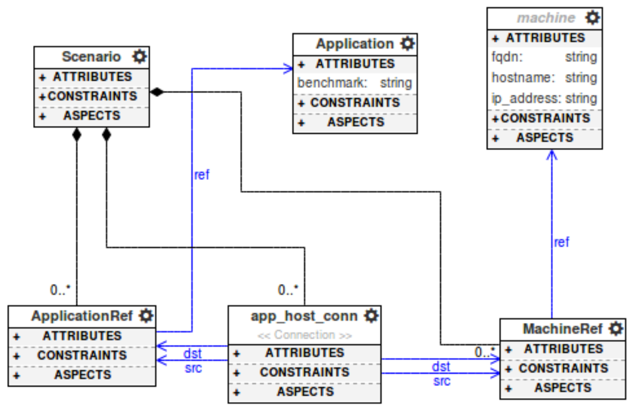

Fig. 6. Snippet of UPSARA Scenario Meta-model

A scenario for an experimentation can involve an application or a group of applications being deployed on a single or a set of runtime platforms. *ApplicationRef* references the *Applications* meta-model. Similarly, *MachineRef* references the *machine* meta-model. As an example, a user might be interested in knowing how an application runs on platform A, and would also like to see the application's performance on platform B. Using the UPSARA language one can easily create these application deployment scenarios by reusing existing artifacts and simply referencing them.

2) UPSARA Specialized Framework Meta-model: To cater to the commonalities and variabilities of the concrete metrics collection frameworks, we provide extensible meta-models with constraints for each such framework that can be specialized from the abstract meta-model. An example of a specialized framework that can be built using the UPSARA DSML is shown in Figure 7. This specialized framework, which we have named as *FECBench*, has been configured to support a subset of monitoring metrics. These include *cpu_util*, *Docker_macro*, *Docker_micro*, *memory_bandwidth_local*, *memory_bandwidth_remote*, and *LLC_bandwidth* system resource metrics. Also, the *fec_viz* object of the type *visualization* meta element is included as a part of this *FECBench*. *fec_viz* includes information about the custom type of visualization that is needed to support analysis for the *FECBench* framework.

This example shows how users create a concrete product line variant from the UPSARA DSML and bring more specialized visualization and analysis aspects as required for each of the custom frameworks. Also, using this meta-model design, the framework can enforce design constraints. For example, it only allows the subset of the metrics as defined by the specialized framework to be available to the end user

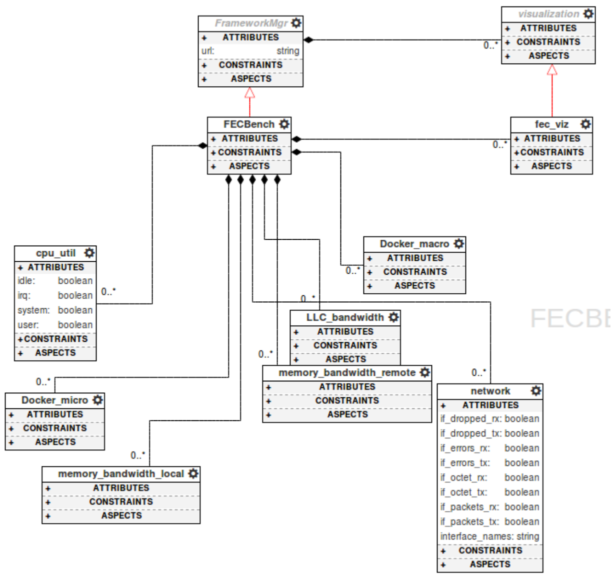

Fig. 7. UPSARA Specialized Framework Meta-model

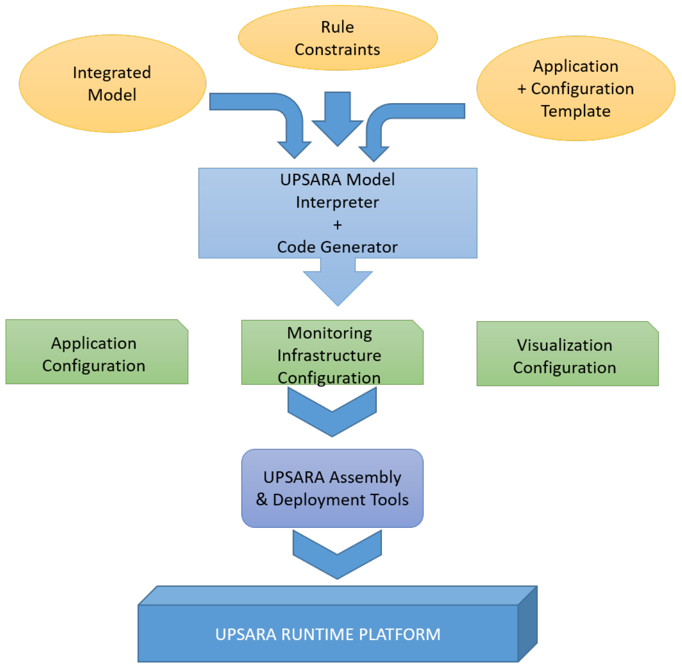
Rule Constraints
Integrated Model
Application + Configuration Template
UPSARA Model Interpreter + Code Generator
Application Configuration
Monitoring Infrastructure Configuration
Visualization Configuration
UPSARA Assembly & Deployment Tools
UPSARA RUNTIME PLATFORM

Fig. 8. UPSARA Generative Process

thereby enforcing constraints and following the *correct-by-construction* principle [14].

IV. UPSARA's Generative Capabilities

UPSARA's generative capabilities are built using the WebGME tool. In WebGME terminology, the model interpreters and the generation can both be handled by writing custom programs called *plugins*. Thus, UPSARA implements the model interpreters and the generators using the *plugins* infrastructure. The *plugins* can be instantiated by users or they can run as service processes. In UPSARA, instantiation of plugins happen when users invoke the plugin by clicking the plugin button in the WebGME environment. Figure 8 shows the process of generation, synthesis and deployment in the UPSARA's generative toolchain.

A. Metric Monitoring Configuration Generation and Provisioning

As described in Section III-A2, a range of metrics collection can be modeled in UPSARA. However, based on the actual underlying metrics collection framework used, only a subset of the metrics are realistically available to the end user at runtime. An example of such a specialized framework is illustrated in Figure 7. Also as shown in Figure 3, there is a mapping between the specialized framework and the platform on which it needs to be deployed. Each metric that needs to be measured can be configured to be monitored from a set of available system monitoring tools. The generative capabilities of UPSARA capture the above relationships between different entities. Moreover, based on the target monitoring tools, UPSARA synthesizes appropriate configuration scripts.

Algorithm 1 depicts the steps involved in the metric configuration generation and deployment. As shown, the generator first needs to traverse the model and get a list of metrics and the hosts on which the metrics are to be configured (Line 2, 3). UPSARA then checks if the selected metric is actually supported by the target runtime platform. This platform capability information is pre-populated and is available to the generator for lookup. If the metric is not supported by the underlying platform, a constraint violation is registered as shown in Line 9. If the metric is supported, we first get the monitoring tool information associated with the metric (Line 10).

Once the appropriate monitoring tool is determined, the associated tool's configuration template is fetched. UPSARA generates an *Ansible* playbook for configuring that metric on the selected cloud platform. UPSARA also generates the configuration script for the monitoring tool to configure the metric selection.

Moreover, as a part of monitoring and analysis, a Jupyter notebook template associated with the framework is loaded and deployed as shown in Line 20. The jupyter notebook files have `.ipynb` extension and comprise a list of `json` objects. The jupyter notebook can be dynamically populated with the metrics parameters that need to be monitored whose information is available in the model.

B. Application Configuration Generation and Orchestration

The process for obtaining application-specific details is similar to the scheme described in Algorithm 1. The generator program first loads the model and gets the information about the applications to be studied. Next it finds the association between the application and the deployment platform on which it needs to be studied. Once the target applications and

Algorithm 1 Metric and Visualization Configuration Generation and Deployment

```
 1: Input ← $(Model)$
 2: $ListofMetrics \leftarrow getMetrics(Model)$
 3: $ListofHosts \leftarrow getPlatformNodes(Model)$
 4: $iframeworkViz \leftarrow getFrameworkViz(Model)$
 5: $mapMetricToHost \leftarrow getMappingofMetricsToHost(Model)$
 6: if $mapMetricToHost! = \emptyset$ then
 7:     for all $V_{m,h} \in mapMetricToHost$ do
 8:         if $isMetricSupportedbyHost(m, h) = False$ then
 9:             Skip                                  ▷ Constraint Violation
10:         $metricTool \leftarrow getMetricTool(Model, m)$
11:         $metricConfTemplate \leftarrow getTemplate(m, metricTool)$
12:                                                   ▷ Generate Artifact
13:         $hostFiles[h].insert(metricConfTemplate)$
14:     for all $host \in hostFiles$ do
15:         for all $files \in hostFiles[host]$ do
16:             $InstantiateDeployment(host, files)$
17:                                                   ▷ Deploy Artifact
18: if $iframeworkViz! = \emptyset$ then
19:     $vizTemplate \leftarrow getVizTemplate(iframeworkViz)$
20:     $deployTemplate(vizTemplate)$                 ▷ Deploy Visualization
```

the runtime platform are determined, UPSARA generates a configuration script. This configuration script is further used by the main framework program which performs the application deployment and execution on the target platform.

V. Meeting the requirements

Based on the description of UPSARA's DSML design and generative capabilities, we now show how UPSARA addresses the challenges and meets the requirements described in Section II-A.

1. Ease of Use: UPSARA provides a visual DSML for easy construction of very large scale experimentation. The performance engineer can also easily and rapidly select metrics of interest which need to be analyzed for the application under test. The generative capabilities in UPSARA create monitoring metric configuration scripts without the need for manual writing of scripts for configuring the monitoring tool.

2. Ensuring the correctness of the configuration: The UPSARA DSML embeds the relationship and constraints between the meta elements of the language. As such these rules ensure that when the user starts designing experiments using WebGME, only the constructs that are supported by the language are available and visible to the user for instantiating. Also, the model interpreter enforces constraint checking which ensures that the experiment that is designed by the user also is checked for constraint violations. This is addressed in Section IV-A.

3. Well-formedness of the generated artifacts: UPSARA provides automated synthesis of application configurations, experimental configurations, metric configurations and the result visualization artifacts. UPSARA's generative process as described in Section IV includes constraint checking rules that detect any violations encountered in the model design. Capturing the violation allows it to ensure that the artifacts generated actually ensure correct functional aspects of the performance measurement study.

4. Extensibilty and tool reuse: Section III-B2 explains how a user can create a specialized framework using existing elements from the UPSARA DSML. It also demonstrated how a user can use existing language elements and build new models using the UPSARA DSML.

5. Support for heterogenous runtime platforms: Section III-B discusses how the UPSARA platform meta-model is able to support heterogeneous runtime platforms. The correct-by-construction generative and deployment capabilities eases the generation and deployment of large-scale configuration artifacts on the heterogeneous runtime platform. This also avoids the accidental complexities involved in configuring such large-scale systems.

VI. Evaluating UPSARA via Use Cases

In this section we use three different use cases and their analyzed performance data to highlight the significant benefits derived from using UPSARA. For our use cases, we have used an experimental setup comprising an Intel Xeon processor whose configuration is listed in Table I. The software details utilized are as follows: the underlying metrics collection frameworks are Collectd (v5.8.0.357.gd77088d), Linux Perf (v4.10.17) and Likwid Perf (v4.3.0). Configuring different monitoring metrics is achieved by utilizing UPSARA. We used applications from the PARSEC [25], SPLASH-2 [25], and DaCaPo [26] benchmarks for the performance analysis.

TABLE I
Hardware & Software Specification of Compute Server

Model Name	Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
Number of CPU cores	16
Memory	32 GB
Operating System	Ubuntu 16.04.3 64-bit

A. Case Study 1- Co-located workload performance analysis

Recent literature indicates that datacenter resources often go underutilized [1]. To increase the utilization, cloud providers tend to consolidate applications by means of overbooking. However, co-location of applications without proper knowledge of the applications' performance profile can result in performance interference, which degrades performance. This occurs due to contention for shared resources. In this use case, we use UPSARA to measure the application's degradation with co-located background workloads.

Our use case application is a computer vision application performing image feature recognition. The image processing application uses Scale Invariant Feature Transform (SIFT) to find the scale and rotation independent features of an image [27]. The application is a server-side application with the clients continuously sending an image of fixed size to be processed. We measure the server-side execution time for processing a single client request. The image processing application is also co-located with other background workloads that share resources on the same host machine.

Figure 9 depicts that the performance of the image processing application deteriorates significantly when co-located with background loads due to varying interference impact. To analyze this variation in the performance of the application, the user utilizes UPSARA to configure the metrics to be monitored, and also to execute the image processing client and server application. Using UPSARA, the user leverages UPSARA's runtime machine learning approaches to determine the dominant metrics that are highly correlated with the application's QoS metric: *execution time*. The figure reveals that L1-iCache-Loads, IPC, IPS, L1-dCache-Loads, LLC-loads, cache-miss are highly correlated with the performance of the application. Using this knowledge, intelligent resource schedulers can be designed, which avoid placing this application with other co-located workloads that are not compute and cache intensive.

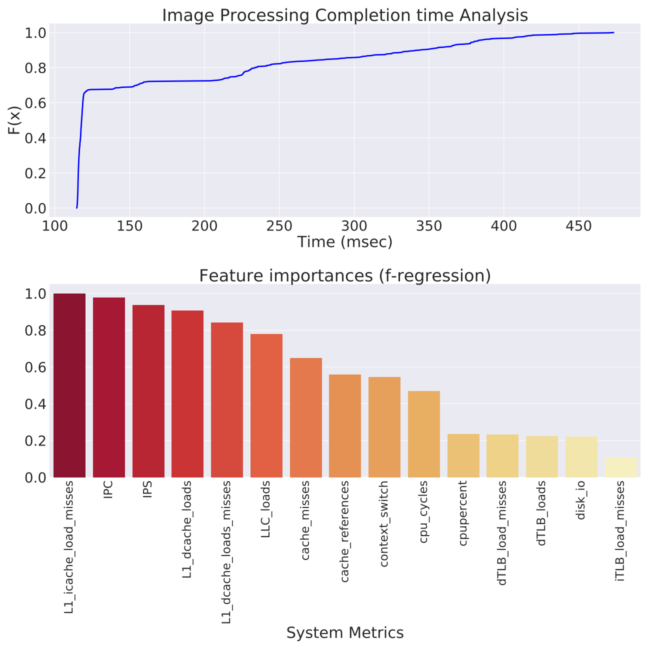

Fig. 9. Performance analysis of image processing service. a) Shows the latency distribution of the completion times of the image processing. b) Shows the dominant measurement metrics that are correlated with the increase in the latency of image processing

B. Case Study 2: Application Resource Utilization Modeling

When deploying applications in the cloud environment, resource management solutions need to allocate sufficient resources to meet application SLOs. The rapid growth of new type workloads such as deep learning and distributed machine learning, are moving to the cloud. As such, studying how different resources are utilized by such workloads becomes critical for cloud/cluster schedulers. As an exemplar, we use the same benchmarks as before for analysis. Our aim was to study the resource utilization of different applications from these benchmarks. Figures 10 and 11 depict the normalized number of system context switches and shared memory bandwidth utilization obtained by orchestrating monitoring components using the UPSARA framework. Due to space constraints we are not displaying rest of the resource utilization metrics. This collected information on resource utilization can be leveraged in building predictive resource utilization models for co-located workloads. Such predictive models can be useful for cloud providers in improving resource allocation algorithms.

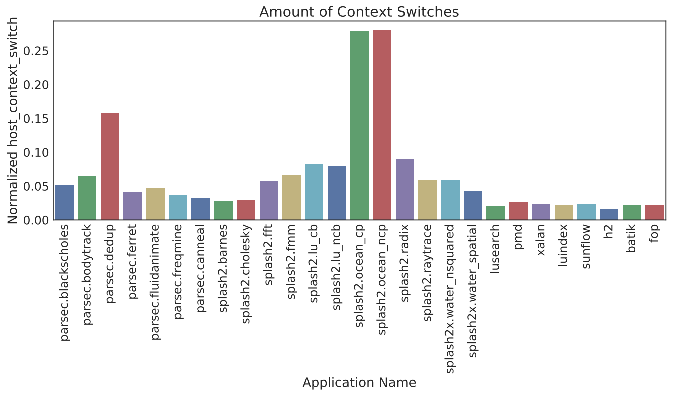

Fig. 10. Normalized amount of context switches imposed by applications from the PARSEC, DaCaPo and Splash-2 benchmarks.

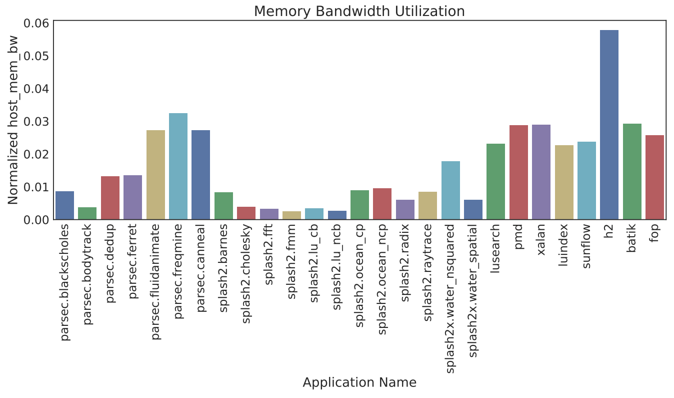

Fig. 11. Normalized memory bandwidth utilization of applications from the PARSEC, DaCaPo and Splash-2 benchmarks.

C. Case Study 3: Studying the impact of Application's Resource configuration

As more and more jobs are migrated to the public cloud, users are faced with a challenge of which configurations to select from a range of virtual machine options provided. The cloud users usually would like to find a cost effective configuration which meets their application's QoS metric. We use UPSARA to measure the application's QoS metric

- *execution time*, for different application container configurations. We measure the execution time for five different CPU core resource configuration options: [1,2,4,8,16] cores. Figure 12 shows the impact of resource configuration on the execution completion times for applications. Using these insights appropriate configuration can be selected such that it satisfies the user requirements.

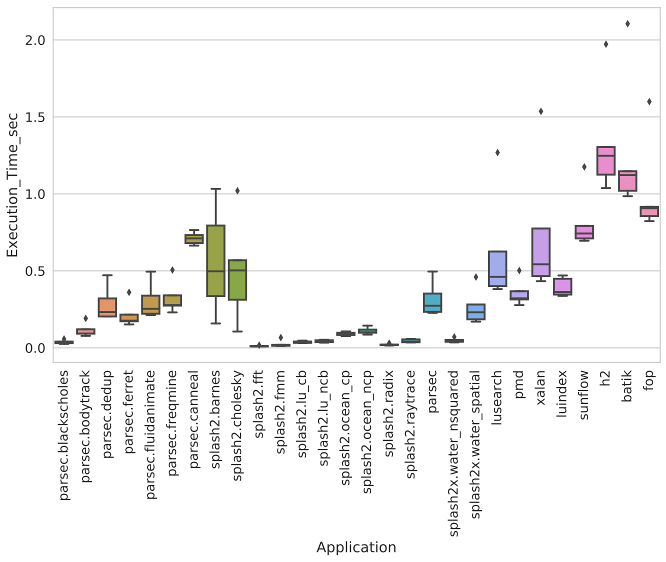

Fig. 12. Impact of different resource configurations on the applications' execution time. Variability in the execution time is due to scaling up in the configuration of the resources assigned to the application.

VII. Related Work

We now compare UPSARA with prior related works. Wienke et. al. [28] have developed a domain-specific language (DSL) for performance profiling, however it is restricted to application profiling only and is coupled to the robotics domain. It utilizes code generation facility to create artifacts required for testing application performance using the Java testing framework.

In [29], a DSL is introduced for web application performance profiling under various load configurations. The DSL automates the cloud resource allocation problem based on the desired QoS goals for the web application. The DSL also generates test cases for load testing the application and monitoring both the system and the application metrics. Similarly, in [30], a DSL called DSLBench is presented. DSLBench generates load testing codes for web application testing. It leverages the Microsoft Visual studio and generates codes in C-Sharp language. AutoPerf [12] presents an automated load testing and resource usage profiling for web service applications. It provides a facility for monitoring resource usage for per request calls in a web session. It also provides a load generation facility.

In [31], an OMG Data Distribution Service-centric performance testing suite is designed leveraging a DSML. It uses model-driven generative capabilities to generate test plans for testing the application's end-to-end QoS under various configurations. It also automates the deployment of the test plans on the cloud environment. Expertus [32] provides an automated performance testing of applications on the cloud environment. It also leverages generative aspects to generate test specification and deployment of the applications using the aspect oriented weaving techniques.

Similar in spirit to our paper, [33] presents a declarative DSL and accompanying model-driven framework for automated end-to-end performance testing of container based micro-services. Similar to application profiling which is the focus of our work, performance testing also involves complex set-up of performance tests, configuration and management of loading infrastructure, deployment under different testing scenarios and post analysis of collected performance data.

While the above DSL tools and methodologies address some of the problems in application performance and system analysis, they still have some drawbacks. In some cases the techniques are tied to a specific programming language, application and/or the runtime platform. As such they cannot be used for heterogeneous runtime platforms and use case scenarios, which UPSARA supports. For example, the solution presented in [31] is tailored towards DDS messaging platform. In [28], performance profiling is geared towards robotic applications running on Java platform. Similarly, web application specific tooling is presented in [29], [30], [12].

Another important differentiation when compared to existing tools is that they do not allow configuring system metrics as captured in the UPSARA DSML. Users still have to write manual configuration scripts for various metric collection probes on the desired runtime platform for performance monitoring. UPSARA's generative, constraint checking and deployment facilities allow for automated synthesis of these monitoring metrics configurations and deployment on the runtime platform.

Moreover, existing tools and approaches lack visual modeling elements which UPSARA's modeling environment provides. This makes configuring of performance monitoring applications much more intuitive thereby presenting a user friendly environment for performance monitoring. UPSARA also creates auto-analysis and visualization notebooks using the Jupyter environment for performance analysis.

VIII. Conclusions

The utility of cloud computing and its services can be significantly improved if performance engineers can rapidly and with ease analyze performance anomalies in cloud-based applications and define appropriate solutions to overcome these problems. Unfortunately, users today face daunting challenges in using existing resource monitoring and application performance modeling frameworks due to the significant variability incurred across these frameworks in terms of APIs, granularity of monitoring, and making sense of the collected data. To that end this paper presents UPSARA, which is an extensible platform based on model-driven engineering technology that has the potential to significantly reduce the entry to barrier for performance monitoring and modeling of applications deployed across the cloud, fog and edge resource systems. UPSARA provides high-level, intuitive abstractions which enable users to quickly set up performance experimentation using visual drag and drop components. Generative and constraint checking components within UPSARA ensure that the generated low-level scripts, such as metric configurations, required for the performance experimentation are correct by construction. The configuration deployment component of the framework, e.g., using Ansible [17], satisfies the deployment of the generated configuration artifacts on the runtime platform. The framework also provides the user with an ability to automate the visualization of results and analysis, which enables users to gain deeper insights into the application performance behaviors.

As future work we plan to extend the DSML to support additional kinds of performance testing usecases, such as the effect of hardware configuration, NUMA bounded placement schemes, cache contention scenarios on application performance. We also plan to support experiment workflows that define dynamic operational patterns as seen in real world situations and analyze application and system performance. These patterns include application workload variability, system failure injections, and collocation application execution scenarios.

UPSARA is available in open source at https://github.com/doc-vu/UPSARA.

Acknowledgments

This work is supported in part by NSF US Ignite CNS 1531079 and AFOSR DDDAS FA9550-18-1-0126 and NIST 70NANB17H274. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or AFOSR or NIST.

References

[1] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 23–33.

[2] S. Shekhar, Y. Barve, and A. Gokhale, "Understanding performance interference benchmarking and application profiling techniques for cloud-hosted latency-sensitive applications," in *Proceedings of the10th International Conference on Utility and Cloud Computing*. ACM, 2017, pp. 187–188.

[3] Y. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, and A. Gokhale, "Poster: Fecbench: An extensible framework for pinpointing sources of performance interference in the cloud-edge resource spectrum." in *Proceedings of the Third ACM/IEEE Symposium on Edge Computing*. ACM, 2018.

[4] "Keras application models," https://keras.io/applications, 2018.

[5] W. Barth, *Nagios: System and network monitoring*. No Starch Press, 2008.

[6] R. Olups, *Zabbix Network Monitoring*. Packt Publishing Ltd, 2016.

[7] F. Forster, "Collectd open source project," http://www.collectd.org, 2017.

[8] "Cache monitoring technology and cache allocation technology," https://github.com/intel/intel-cmt-cat, 2018.

[9] C. Heger, A. van Hoorn, M. Mann, and D. Okanović, "Application performance management: State of the art and challenges for the future," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 429–432.

[10] J. Walter, S. Eismann, J. Grohmann, D. Okanovic, and S. Kounev, "Tools for declarative performance engineering," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 53–56.

[11] N. Michael, N. Ramannavar, Y. Shen, S. Patil, and J.-L. Sung, "Cloudperf: A performance test framework for distributed and dynamic multi-tenant environments," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 189–200.

[12] V. Apte, T. Viswanath, D. Gawali, A. Kommireddy, and A. Gupta, "Autoperf: Automated load testing and resource usage profiling of multi-tier internet applications," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 115–126.

[13] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[14] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[15] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918–2933, 2014.

[16] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-oriented Software Architecture: on patterns and pattern language*. John wiley & sons, 2007, vol. 5.

[17] "Ansible-automation for everyone," https://www.ansible.com/, 2018.

[18] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *ELPUB*, 2016, pp. 87–90.

[19] M. Maróti, R. Kereskényi, T. Kecskés, P. Völgyesi, and A. Lédeczi, "Online collaborative environment for designing complex computational systems," *Procedia Computer Science*, vol. 29, pp. 2432–2441, 2014.

[20] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "(wip) cloudcamp: Automating the deployment and management of cloud services," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 237–240.

[21] Y. D. Barve, P. Patil, A. Bhattacharjee, and A. Gokhale, "Pads: Design and implementation of a cloud-based, immersive learning environment for distributed systems algorithms," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 1, pp. 20–31, 2018.

[22] Y. D. Barve, P. Patil, and A. Gokhale, "A cloud-based immersive learning environment for distributed systems algorithms," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 1. IEEE, 2016, pp. 754–763.

[23] H. Cho, J. Gray, and E. Syriani, "Creating visual domain-specific modeling languages from end-user demonstration," in *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*. IEEE, 2012, pp. 22–28.

[24] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[25] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 47–56.

[26] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, no. 10. ACM, 2006, pp. 169–190.

[27] M. Sonka, V. Hlavac, and R. Boyle, *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[28] J. Wienke, D. Wigand, N. Koster, and S. Wrede, "Model-based performance testing for robotics software components," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2018, pp. 25–32.

[29] Y. Sun, J. White, S. Eade, and D. C. Schmidt, "Roar: A qos-oriented modeling framework for automated cloud resource allocation and optimization," *Journal of Systems and Software*, vol. 116, pp. 146–161, 2016.

[30] N. B. Bui, L. Zhu, I. Gorton, and Y. Liu, "Benchmark generation using domain specific modeling," in *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*. IEEE, 2007, pp. 169–180.

[31] K. An, T. Kuroda, A. Gokhale, S. Tambe, and A. Sorbini, "Model-driven generative framework for automated omg dds performance testing in the cloud," *ACM Sigplan Notices*, vol. 49, no. 3, pp. 179–182, 2014.

[32] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu, "Expertus: A generator approach to automate performance testing in iaas clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 115–122.

[33] V. Ferme and C. Pautasso, "A declarative approach for performance tests execution in continuous software development environments," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 261–272.