Towards a Framework for Building SaaS Applications Operating in Diverse and Dynamic Environments

Ashish Agrawal(✉) and T.V. Prabhakar

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, Kanpur, UP 208016, India
{agrawala,tvp}@cse.iitk.ac.in

Abstract. Enterprises have increasingly adopted the Software-as-a-service (SaaS) model to facilitate on-demand delivery of software applications. A SaaS customer - tenant - may operate in diverse environments and may demand a different level of qualities from the application. A tenant may also operate in a dynamic environment where expectations from the application may change at run-time. To be able to operate in such environments, SaaS application requires support at both the architecture and implementation levels. This paper highlights the issues in building a SaaS that can accommodate such diverse and dynamic environments. We propose a methodological framework called *Chameleonic-SaaS* that abstracts out the responsibilities involved and provides guidelines to realize it. Our framework introduces variability in the architecture to manipulate the architecture-level decisions, especially tactics. Feasibility of the framework is demonstrated by an example of a MOOC application.

Keywords: Software as a service · Variability · Adaptive SaaS · Dynamic quality attributes

1 Introduction

Software-as-a-Service (SaaS) - a delivery model for software applications - attracts customers by presenting features such as no up-front cost, on-demand provisioning at an application-level of granularity and free from maintenance [3,12]. In SaaS model, the service provider is responsible for managing all service components (software and hardware) and ensuring application-level quality attributes desired by a customer. These SaaS customers - "tenants" - may operate in diverse environments and may demand different levels of qualities (e.g., low or high availability) from the application [4,15]. For example, considering an ERP SaaS, a small organization may need low availability (95 %) and an enterprise may demand high availability (99.99 %). Similarly, a tenant may also operate in a dynamic environment where expectations from the application may change at run-time to accommodate changes in the environment. In our scenario, the small organization may desire to have high availability for a time

DOI: 10.1007/978-3-319-48992-6_22

period such as peak load and business events. The motivation behind the need for such dynamic quality requirements is the fact that some quality attributes have an impact on the operational cost of the application, and the application may not require high values of these quality attributes all the time. For example, if an application achieves high availability by replicating to a redundant server, this additional server will increase the operational cost. Figure 1 depicts a case of quality expectations of tenants of a SaaS.

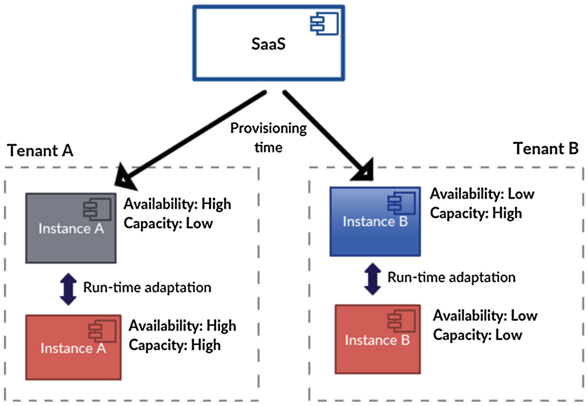


Fig. 1. An example showing diverse and dynamic quality expectations from a SaaS

To make the offering attractive to the tenants, a SaaS should have the ability to address diverse and dynamic quality requirements. From an architectural perspective, two most common patterns [4,14] for building a SaaS are; (1) *Multi-tenant* where all tenants share a common instance along with the code components, and (2) *Multi-instance* where every tenant has a dedicated instance allocated to it.

For building a SaaS operating in diverse and dynamic environments, *Multi-tenancy* would be beneficial in terms of operational cost and maintenance. However, this pattern requires designing tenant-aware components that can increase development cost and time to market. Although the development cost would be high, it might be compensated by lower operational cost [6]. Contrary to this, benefits of *Multi-instance* are; less time to market, lower design cost, and flexibility for customization. However, *Multi-instance* pattern may have high operational cost and high maintenance if there are a large number of tenants. A service provider can select a pattern by analyzing these parameters in the context of its business goals and policies. One can also use a combination of these patterns where a group of tenants shares a common instance.

One thing to note here is that addressing diversity issues of the tenants in *Multi-tenancy* may create a very complex architecture and design that can create issues for maintaining the service. In some scenarios, it may be easy to maintain multiple simple instances than a single complex instance. In this paper, we focus on using the *Multi-instance* pattern for implementing a SaaS as it provides more flexibility to handle diverse and dynamic environments.

One way to accommodate diversity and dynamism is to identify the set of possible quality requirements and build different versions of the application separately for every member of such a set. However, in this case, development and maintenance cost would be very high. As the quality requirements are also dynamic in nature, migrating between members of the set might not be possible. Another approach could be to provide maximum values of quality attributes to all tenants at all time. However, this may not be cost-effective from the provider's perspective. Diversity and dynamism can also be handled by customizing the size of an instance (e.g., CPU, RAM, etc.) according to a tenant's requirements. However, only a few quality attributes (e.g., performance, capacity, etc.) can be changed using this approach. Also, variations in quality values will be limited.

Our idea for solving these issues is to model quality attributes as scriptable resources. That means that the application exposes a programmable interface to the tenants for requesting quality attributes. To handle diversity, a tenant can specify its quality requirements at the time of requesting a new instance i.e. provisioning-time. To accommodate dynamic environments, a tenant can change the quality attributes of its instance dynamically at run-time on demand basis. To handle such requests from a tenant, application modifies the architecture-level decisions of its instance such as tactics. To facilitate such features and to manage all running instances, the application should be designed with the ability to dynamically modify its architecture.

To realize our approach, we identify the suitable tactics and introduce variability in the architecture by modeling these tactics as variation points. To be able to change the quality attributes dynamically, we model the service as an adaptive system using the concepts of MAPE-K loop architecture [10]. Findings of our investigation are formulated as a methodological framework called *Chameleonic-SaaS*. Main contributions of this paper are:

- An idea to model quality attributes as scriptable resources.
- A methodological framework called *Chameleonic-SaaS* for building SaaS operating in diverse and dynamic environments. The framework abstracts out the responsibilities involved and provides guidelines for the same.

The rest of the paper is organized as follows. Section 2 describes the problem statement and our approach. Section 3 explains the *Chameleonic-SaaS* framework in detail. Section 4 presents an example by building a MOOC application. Section 5 provides a brief summary of existing work related to this paper. Section 6 discusses benefits and limitations of our approach. Section 7 concludes the paper with scope of future work.

2 Problem Statement and Approach

This section defines the problem statement and describes our approach.

2.1 Problem Statement

This work aims to investigate the issues in building a SaaS operating in diverse and dynamic environments. Requirements from such a SaaS are:

- Service should have the ability to address diverse quality requirements of different tenants at provisioning-time.
- Service should have the ability to change quality attributes for a particular tenant dynamically at run-time.
- It should be easy to maintain the service.

2.2 Approach

Our idea is to expose quality attributes as scriptable resources to the tenants of a SaaS. Using such resources, a tenant can customize the set of quality attribute values provided by an application instance. Such customization can occur either at provisioning-time or dynamically at run-time on demand basis. Here, customizations in the quality attribute values are achieved by modifying architecture-level decisions of the application.

This leads to the question of what architectural decisions need to be changed in the architecture. Such architectural decisions should only impact quality attributes of the application. We use the architectural tactics as the architectural decisions that can be modified at run-time. A tactic is an architectural tool that can be used to improve a particular quality attribute of an application [2]. For example, *Ping & Echo* [2] is a tactic to improve the availability of an application by detecting failures such as network failure. Thus, to modify quality attributes of a tenant's instance, SaaS can add or remove tactics in its architecture. The approach mentioned above leads to a natural question:

- **RQ1**: How to externally add a tactic to an application instance deployed on a virtual machine?

In architecture, realization of a tactic can be seen as a set of operations (add, remove or modify) on architectural elements – components, connectors, and links. We categorize the architectural elements into three groups; (1) *pure application elements* incorporating application logic, (2) *pure tactic elements* which are tactic elements that are independent of application logic, and (3) *application-specific tactic elements* which are tactic elements that requires knowledge of application logic. For example, *Ping & Echo* tactic can be implemented using three pure tactic components; *PingSender* sends ping requests periodically, *PingReceiver* sends an echo for a received ping, and *Monitor* notifies the occurrence of a failure.

One of the ways to achieve *Ping & Echo* tactic at a virtual machine level is as follows. Deploy *PingReceiver* on the virtual machine that host the application and deploy rest of the tactic components on a separate virtual machine. As there are no links between the tactic components and the application components, we can add this tactic without any modification to the application components.

Some tactics may include application-specific components. For example, implementation of the *Passive Redundancy* tactic requires a *StateManager* component to fetch and update the application state. The *StateManager* being an application-specific component, has to be a part of the application architecture. If the application exposes an interface for the *StateManager* with dynamic binding then the *Passive Redundancy* tactic can be added to the application.

To be able to add a tactic, application architecture should provide support by exposing application-specific tactic elements. This decision to implement such elements in the application is based on the trade-off between customization enabled by the tactic and its impact on the development cost. By examining such support, we can decide whether it is feasible to use a tactic in the present context. An application-specific tactic component may also enable multiple tactics. For example, *StateManager* may enable both *Passive Redundancy* and *Rollback*. These components have a higher impact on the ability to customize in comparison with the development cost.

Not all tactics can be added using this approach. For example, tactics related to quality attributes not discernible at run-time cannot be used here. Similarly, if the application architecture does not provide support by exposing application-specific tactic components and the ability to run-time binding, it may not be possible to add such tactics. The capability of our approach to change quality attributes depends on the number of tactics supported by the application architecture for dynamic addition.

These tactic-specific components may have an adverse effect on other quality attributes. In the case of a large number of such components, they should be incorporated in only the instances whose tenants demand variations in the respective qualities. Some tactic components may also have an exclusive relationship with other tactic components. Thus, to maintain the system easily and dynamically select the tactics components, we introduce variability in the application architecture where tactic components are modeled as variation points.

3 Chameleonic-SaaS Framework

Findings of our investigation on building SaaS applications for diverse and dynamic environments are formulated as a methodological framework called *Chameleonic-SaaS*. Applications built using this framework can provision instances with different quality attributes to address diverse quality requirements of SaaS-users. Quality attributes of such instances can also be changed dynamically at run-time, to accommodate dynamic operating environments. This framework abstracts out the responsibilities involved and provides architectural guidelines for building such SaaS applications. Steps of the framework (depicted in Fig. 2) are explained in the following sections.

3.1 Identify QA Scenarios

The first step is to identify the quality requirements of the application that can vary either at provisioning time or run-time. This task has to be done manually

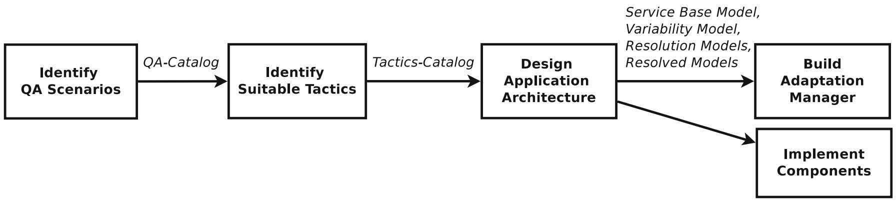

Identify QA Scenarios
QA-Catalog
Identify Suitable Tactics
Tactics-Catalog
Design Application Architecture
Service Base Model, Variability Model, Resolution Models, Resolved Models
Build Adaptation Manager
Implement Components

Fig. 2. Steps of the Chameleonic-SaaS framework

by analyzing the application requirements (using requirement specification or user stories) and separate out such quality concerns. By this analysis, we identify a list of quality attributes that can differ in multiple instances of the application or can vary at run-time in a particular instance. For example, in a SaaS application, availability requirements may vary with tenants from highly available to moderate available. Similarly, a tenant having moderate availability initially, may require high availability on an environmental change (e.g., peak load, business event, etc.). The output of this step is a *QA-Catalog* that includes; (1) Quality attributes identified by the analysis along with desired range of their values, and (2) Scenarios for run-time variation in the quality attributes documented as *Quality Attribute Scenarios (QASs)* [2].

3.2 Identify Suitable Tactics

In this step, we identify the architectural tactics that can be used to achieve the desired quality requirements specified in the *QA-Catalog*. This task is done by analyzing the tactics repositories [2,16] along with the application architecture using the methodology specified in Sect. 2.2. Tactics may also have dependencies with each other. For example, *Active Redundancy* and *Passive Redundancy* tactics have an exclusive relationship with each other and cannot be applied together in a system. Similarly, a tactic for one quality attribute may also have an impact on other quality attributes. For example, *Active Redundancy* tactic of availability can have an adverse impact on performance. This step aims to identify a set of feasible tactics for every QAS specified in the *QA-Catalog* by analyzing the architectural tactics, their relationships with each other and their impact on the quality attributes. The output of this step is an artifact called *Tactics-Catalog* that consists a list of mappings between a quality value and a set of tactics that can be used to achieve that quality.

3.3 Design Application Architecture

In the previous step, we identified a set of tactics that can be incorporated into the architecture to handle the desired QASs. These tactics need to be incorporated in the architecture design in a way such that their existence can vary with tenants as well as with time for a particular tenant. Instead of designing multiple architectures of the application, our approach is to introduce variability in the architecture for quality concerns. To model variability, we follow the *Orthogonal Variability Modeling (OVM)* [13] approach and model the quality concerns separately from the application base architecture. Following the OVM approach, we use the *Common Variability Language (CVL)* [8] to describe the variability. CVL is a domain independent language for specifying variability models and has an execution engine to generate the *resolved models* i.e. instance architectures. In this step, we prepare the following models as depicted in Fig. 3. Examples of these models in our context are presented in the Sect. 4.

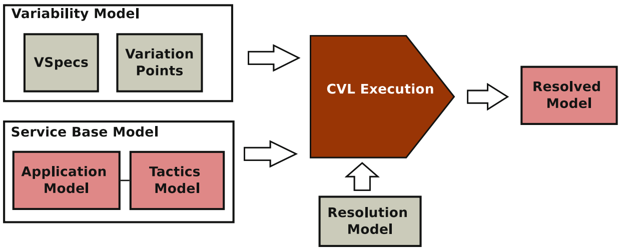


Fig. 3. Models in the CVL approach

1. **Application Model**: This model captures the architectural elements that are common to all tenants. Our approach is to capture quality related concerns in a separate model and this model only includes the support required to handle those concerns. This approach gives us the ability to re-use such quality related concerns and manage them independently from the application components. Thus, the *Application Model* includes *pure application elements* and *application-specific tactics elements* (defined in Sect. 2.2). The model can be described using any domain specific language such as UML.
2. **Tactics Model**: This model includes *pure tactic elements* that are agnostic to the application logic.
3. **Service Base Model**: In order to describe variability, this model is prepared by combining the *Application Model* and the *Tactics Model*. Apart from the elements of these two models, this model also includes elements that establish links between them. This model is considered as a base model to describe variability.
4. **Variability Specification (VSpecs)**: This model specifies the variability at an abstract level i.e. irrespective of its mapping to the *Service Base Model*. We incorporate the quality attributes and the tactics as first-class concepts in this model. This approach provides us the ability to choose variations at the granularity of tactics. Thus, the *QA-Catalog* and the *Tactics-Catalog* are used to describe tactics and relationships between tactics. Such relationships can be modeled as *choice multiplicity* or constraints. For example, a tactic for fault recovery requires a tactic from fault detection. These variations are captured as a tree structure of choices.

5. **Variation Points**: This model includes variation points referencing to the *Service Base Model.* Variation points are the modifications applied to the *Service Base Model* to generate an instance architecture. For example, a variation point specifies the existence of a tactic component called *PingSender.* To re-use the variation points related to pure tactics elements, they can be combined and represented as *Configurable Units.* Every variation point has a binding to exactly one VSpec.
6. **Resolution Models**: This model resolves VSpecs. For example, a choice resolution may resolve a choice VSpec. In our case, every QAS is mapped to a resolution model representing the tactics to be selected for that QAS. Thus, *QA-Catalog* is used to generate different resolution models corresponding to each QAS. These models are used to generate architectures of the instances i.e. *Resolved models.*

3.4 Implement Components

In our scenario, tactic components are dynamically added to an existing instance at run-time. Such operation requires modification in the connections between application and tactics components. Thus, they should be implemented in such a way so that their binding can be configured at run-time. Some techniques that can be used for such implementation are:

– *Encapsulate*: Components should provide an explicit interface such as an API.
– *Defer Binding*: Components should defer their binding so that it can be decided or changed at run-time.

3.5 Build Adaptation Manager

This component is responsible for managing the SaaS application (and its instances) for adaptation at provisioning time and at run-time. It exposes an interface to the tenants for two kinds of operations; (1) Provisioning of an instance for a given set of quality requirements, and (2) Provisioning of a quality attribute value to an existing instance. Design of the *Adaptation Manager* is based on the MAPE-K loop [10] of adaptive systems. Figure 4 depicts run-time view of a SaaS application that includes the following components:

– **Event Monitor**: This component is responsible for capturing the events that demands provisioning of instances or quality attributes. Sensors running on an instance to monitor its environment can trigger the *Event Monitor.*
– **Architecture Analyzer**: On an adaptation request from the *Event Monitor*, this component identifies the QAS from *QA-Catalog* and analyzes the current architecture of the concerned instance (stored in *Architectural Knowledge Repository*) to check the feasibility of the requested operation.
– **Adaptation Planner**: On the occurrence of a QAS, *Planner* component identifies the desired tactics from the *Tactic-Catalog* and generate instance architecture using a *Resolution Model.* Using the current architecture of the concerned instance, it plans for changes to be applied to the instance.

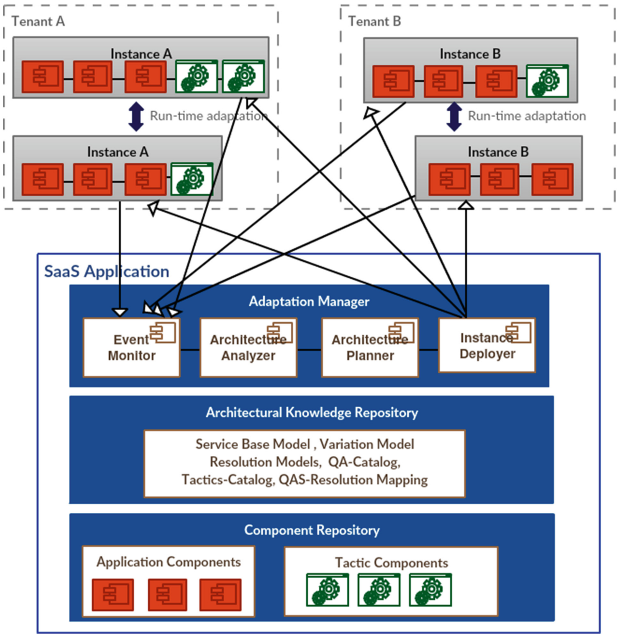


Fig. 4. A run-time scenario of a SaaS application

- **Instance Deployer**: This component executes the changes proposed by the *Planner* component. To deploy the tactics related component, it uses the programmable interfaces of underlying cloud resources.
- **Architectural Knowledge Repository**: This repository contains the architectural knowledge that is used by other components of the *Adaptation Manager* such as *Application Model*, *Tactic Model*, *Variability Model*, *QA-Catalog*, *Tactics-Catalog* and current architecture of all instances (*Resolution Models*).

4 Example

This section presents an example SaaS called *MOOC Management System (MMS)* built using the methodology specified by the *Chameleonic-SaaS* framework. This service facilitates provisioning of application instances to customers (organizations or individuals) to deliver and manage online courses. Quality attributes desired by a MMS instance such as capacity, availability and performance may vary with the organizations depending on the factors such as the number of students and credit vs. non-credit courses. For a particular organization, quality expectations may also change during run-time on the occurrence of events such as quizzes/exams and real-time hangout sessions. This study aims to check the applicability of our approach by identifying QASs and tactics, designing application architecture and deploying the service.

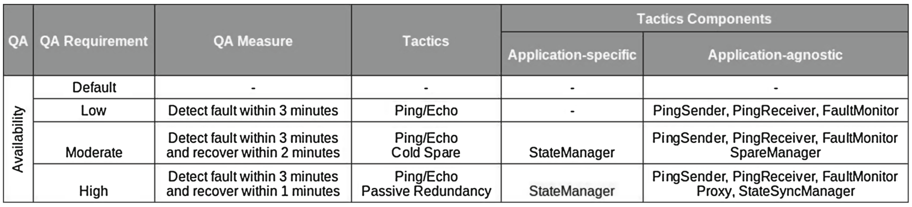
In this study, we focus on the availability quality attribute of the MMS. Availability is considered as an expensive quality attribute as its realization through redundant resources increases operational cost. To make the offering attractive to the customers, MMS facilitates customization of availability values at provisioning time as well as at run-time. We categorize the range of availability values offered by the service into four types; (1) Default availability (no additional support) (2) Low availability, (3) Moderate availability, and (4) High availability. During provisioning of a new instance, service creates an appropriate instance architecture according to the desired requirements of the tenant. MMS also exposes a programmable interface to change availability of a provisioned instance. Projecting availability as scriptable resources enables the tenant to become cost-efficient by dynamically varying between the different availability offerings.

Availability can be provisioned to an instance either on a direct request from the customer or on the occurrence of an event in the application environment. We identified four QASs that may demand variations in the availability values of a running instance. The basic idea of these QASs is that in normal operations, the application works with low availability values and additional availability is provisioned only when these is a demand for the same. These QASs are:

– **QAS-1**: "During a quiz period, the application has high availability". As quizzes/exams have time duration associated with them, the application is expected to have high availability to avoid or at least reduce any downtime.
– **QAS-2**: "If new course material is released, the application has moderate availability". It has been observed that release of course material (stimulus) results in a large number of students accessing the application. Downtime during such periods should be avoided. However, it is not as critical as quizzes/exams.
– **QAS-3**: "In normal operations, the application has low availability". In the absence of any critical events, the application is expected to have low availability.
– **QAS-4**: "If the course is migrated to read-only (self-paced) mode, the application has default availability". The self-paced mode is a low priority scenario, and the application does not need any additional support for availability (low availability) to have minimum operational cost.

In our example, these requirements are handled by realization of three tactics; *Ping & Echo*, *Cold Spare* and *Passive Redundancy* [2]. Figure 5 depicts mapping of these tactics to their respective quality requirements along with the tactics components used to realize them in the application. Realization of *Cold Spare* and *Passive Redundancy* requires application to expose an application-specific tactic component called *State Manager* to be able to get and set application state.

Figure 6 depicts the *Service Base Model* along with variation points bound to the *VSpecs*. The *Service Base Model* is prepared by integrating the *Application Model* and the *Tactic Model*. In the *VSpecs*, availability is modeled as an optional



QA	QA Requirement	QA Measure	Tactics	Tactics Components	
				Application-specific	Application-agnostic
Availability	Default	-	-	-	-
	Low	Detect fault within 3 minutes	Ping/Echo	-	PingSender, PingReceiver, FaultMonitor
	Moderate	Detect fault within 3 minutes and recover within 2 minutes	Ping/Echo Cold Spare	StateManager	PingSender, PingReceiver, FaultMonitor SpareManager
	High	Detect fault within 3 minutes and recover within 1 minutes	Ping/Echo Passive Redundancy	StateManager	PingSender, PingReceiver, FaultMonitor Proxy, StateSyncManager

Fig. 5. Availability requirements and the corresponding tactics

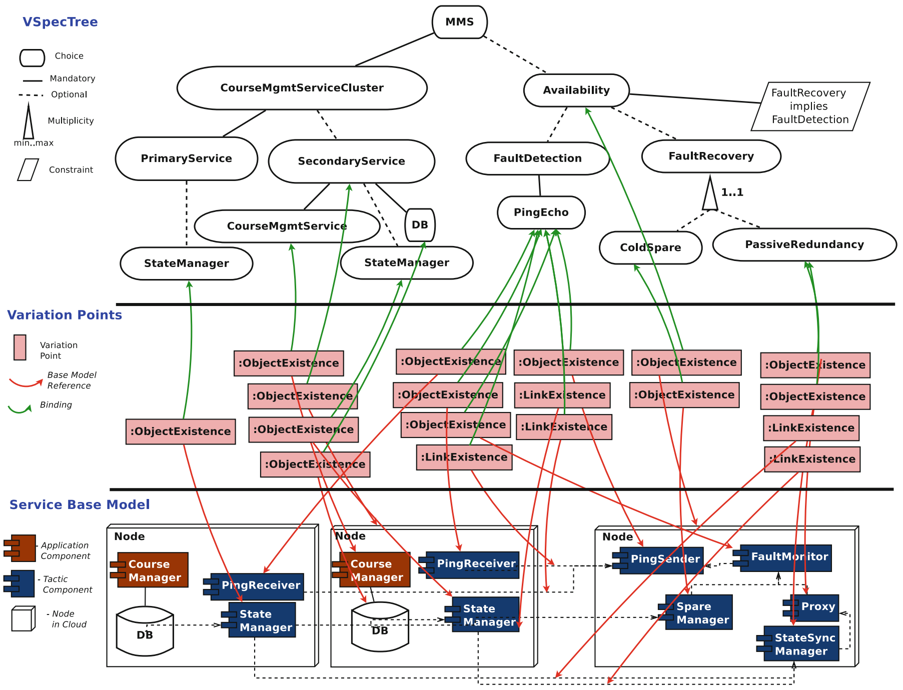


Fig. 6. Service base model with variation points bound to the VSpecs

choice that further has two child choices; *FaultDetection* and *FaultRecovery*. There is also a constraint specifying that *FaultRecovery* requires *FaultDetection* to be present in the instance. *FaultDetection* has *PingEcho* as a child choice that is linked to various *Variation Points* relating to the existence of components (*PingSender*, *PingReceiver*, etc.) and links. Figure 7 depicts resolution model for QAS-1 where *PassiveRedundancy* choice is *True* but *ColdSpare* is *False*. Figure 8 depicts architectures of the various instances generated by the service depending upon the resolution models.

For variation triggered by the events in the application environment, a sensor to monitor events - course material release, quiz period and self-pace mode - is implemented in the application that triggers the *Event Monitor* component of

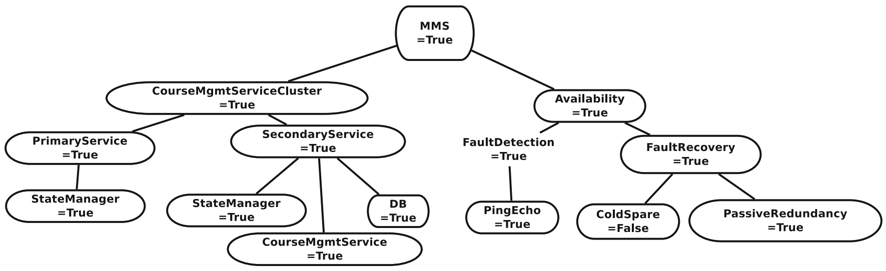


Fig. 7. Resolution model for QAS-1 (PingEcho and PassiveRedundancy)

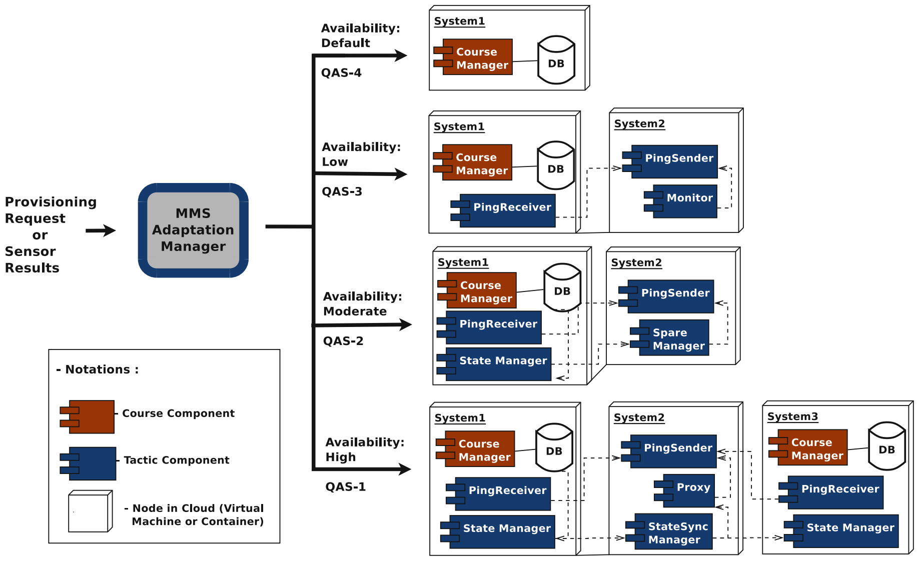


Fig. 8. Architectures generated by the *Adaptation Manager* depending on the QAS

the *Adaptation Manager*. These events are analyzed to check occurrence of any QAS.*Adaptation Manager* also exposes an API through which a customer can directly request for an availability value (default, low, moderate, or high) to an existing instance. Depending upon the current architecture of the instance and the desired QAS, *Adaptation Manager* modifies the instance architecture by adding or removing components.

Figure 9 presents experiments results conducted by dynamically provisioning the availability values to a MMS instance. In our setup, service is offered by creating MMS instance over Linux containers (LXC). LXC containers were setup on a virtual machine (1CPU Core, 2 GB RAM) running Ubuntu operating system. For deployment of tactics components, we used the *Puppet* tool [18]. The results show that adding quality to an existing instance is fast due to quick creation of containers. Also, *Passive Redundancy* has less fault recovery time compared to *Cold Spare* tactic as the later requires creating a new container to recover. These timings directly depend on our execution environment and should not be used as benchmarks.

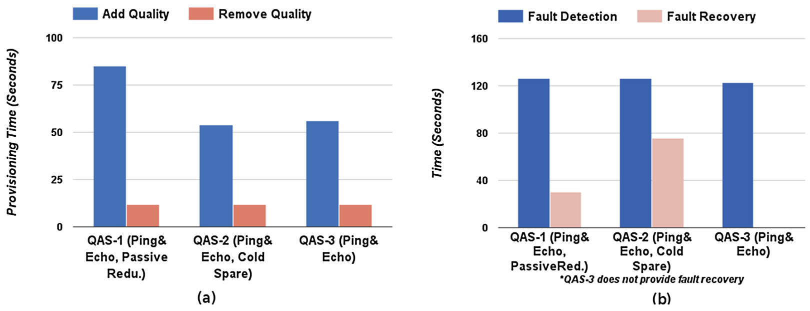


Fig. 9. Experiment results for availability scenarios in MMS (a) Provisioning time in adding or removing the QASs, and (b) Availability benefits in terms of time consumed in fault detection and fault recovery

5 Related Work

The demand for tenant-specific customization of a service has been highlighted by several researchers [1,12,17]. Here, customization is desired in features, work-flow, user-interface, etc., and facilitated using the virtualization techniques [4]. In context of SaaS applications, researchers have identified some architectural patterns such as *Multi-tenancy* and *Multi-instance* and discussed their impact on the quality attributes [4,6,14]. Koziolek [11] discussed various quality requirements from a SaaS such as resource sharing, scalability, maintainability, customizability, and usability. The work also includes an architectural style called SPOSAD based on multi-tier style. Software engineering issues with developing SaaS applications have also been discussed [5].

Variability has been presented as a quality attribute of architecture [2] and has been extensively used in Product Line Engineering (PLE). However, variability in quality attributes (performance variability, availability variability, etc.) has not been much used and requires more explorations [7].

Several researchers have proposed techniques to design a SaaS as a *Product Line Architecture* by introducing variability in the architecture [1,15,17]. Matar et al. [1] discussed different kinds of variability for a SaaS such as application variability, business process variability and provisioning variability. However, most of these works are focused only on the variations in the feature models. These approaches are also not able to handle the environmental changes demanding variations only in quality attributes. Horcas et al. [9] presented a technique to inject functional quality attributes (that results in functional components) in an application. In our work, our focus is on varying only the quality attributes of a SaaS instance, by changing architectural decisions at a tactic level granularity.

6 Discussion

Quality attributes exposed as scriptable resources enable variation in their values for a running instance. As run-time quality attributes have an impact on the operational cost of the instance, a tenant can exploit such resources to achieve cost-efficiency by dynamically migrating between different offerings of quality attributes on demand basis.

Modeling quality related concerns separately from the functional concerns provides reusability of the quality concerns across multiple applications, and modifiability of these concerns. For example, *Tactics Model* can be shared between multiple applications. Similarly, in our MMS application, we can add a new tactic such as *Rollback* without modifying the *Application Model* as the support required by this new tactic (*StateManager*) is already exposed by the *Application Model*. In our approach, all instances of the SaaS are generated using a single architecture which makes the maintenance easier compared to the approach where every instance is designed and build separately.

In our framework, tactics are modeled as first-class concepts in the *Variability Model*. As tactics are standard validated tools to improve quality attributes, such modeling helps in evaluating the variations in an instance architecture in terms of their impact on the quality attributes.

The framework only considers variations in the architectural decisions of an instance, and does not cover other decisions such as deployment-level decisions (e.g., sizing of hardware resources, etc.), implementation-level details (e.g., code, logging, etc.), or application functionality. We do not aim to replace the other techniques but to augment their capability to reach more diverse levels of quality attributes. In a holistic approach, variability at different levels (architecture, deployment, implementation, features) can be combined.

Another limitation of our work is that we presented a methodological framework where several steps of the framework like *Identify Tactics*, merging the *Application Model* with the *Tactic Model*, etc. are not automated. In this paper, we explored adding tactics at the top level of application architecture. However, variations may be desired at a lower level architecture element such as a microservice. Our framework can be further extended to handle such scenarios.

Not all quality attributes can be modeled as scriptable resources. For example, quality attributes not discernible at run-time such as modifiability cannot be changed using our approach. The capability of our approach to change quality attributes depends on the number of tactics supported by the application architecture for dynamic addition (in terms of application-specific tactic components exposed by the application). Our approach has an impact on design and development cost of the application. Re-using the tactics related concerns can help in reducing such overhead.

7 Conclusion and Future Work

In this paper, we presented an approach to offer quality attributes of a SaaS application as scriptable resources. To build such an application, we need to identify the suitable tactic-level architectural decisions, introduce variability in the architecture and incorporate the ability to change an instance architecture dynamically at run-time. In our methodological framework, tactics are modeled as first-class concepts in the application architecture. This enables to articulate the impact of architectural variations on the quality attributes. Our example MOOC application facilitates a tenant to vary its availability values between default, low, moderate, and high. The current version of the *Chameleonic-SaaS* framework is applicable only for a multi-instance SaaS. To build a multi-tenant SaaS with the ability to dynamically change quality attributes of tenants can be further explored. To transform an existing application to a SaaS would also be an interesting problem especially in the scenarios when the application does not provide any direct support for adding the tactics externally.

Acknowledgment. The authors gratefully acknowledge the financial support from Tata Consultancy Services and MHRD, Govt. of India for this work.

References

1. Abu Matar, M., Mizouni, R., Alzahmi, S.: Towards software product lines based cloud architectures. In: 2014 IEEE International Conference on Cloud Engineering (IC2E), pp. 117–126, March 2014
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley Professional, Boston (2012)
3. Benlian, A., Hess, T.: Opportunities and risks of software-as-a-service: Findings from a survey of IT executives. Decis. Support Syst. **52**(1), 232–246 (2011)
4. Bezemer, C.P., Zaidman, A.: Multi-tenant SaaS applications: maintenance dream or nightmare? In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL 2010, pp. 88–92. ACM, New York (2010)
5. Cai, H., Wang, N., Zhou, M.J.: A transparent approach of enabling saas multi-tenancy in the cloud. In: 2010 6th World Congress on Services (SERVICES-1), pp. 40–47, July 2010
6. Frederick Chong, G.C., Wolter, R.: Multi-tenant data architecture, June 2006. http://msdn.microsoft.com/en-us/library/aa479086.aspx
7. Galster, M.: Architecting for variability in quality attributes of software systems. In: Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW 2015, pp. 23:1–23:4. ACM, New York (2015)
8. Haugen, O., et al.: Common variability language (CVL). OMG Submission (2012)
9. Horcas, J.M., Pinto, M., Fuentes, L.: Injecting quality attributes into software architectures with the common variability language. In: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE 2014, pp. 35–44. ACM, New York (2014)
10. Jacob, B., Lanyon-Hogg, R., Nadgir, D.K., Yassin, A.F.: A practical guide to the to the IBM autonomic computing toolkit, April 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246635.pdf
11. Koziolek, H.: The sposad architectural style for multi-tenant software applications. In: 2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 320–327, June 2011

12. La, H.J., Kim, S.D.: A systematic process for developing high quality SaaS cloud services. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 278–289. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10665-1_25
13. Metzger, A., Pohl, K.: Software product line engineering and variability management: achievements and challenges. In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 70–84. ACM, New York (2014)
14. Mietzner, R., Unger, T., Titze, R., Leymann, F.: Combining different multi-tenancy patterns in service-oriented applications. In: Proceedings of the 13th IEEE International Conference on Enterprise Distributed Object Computing, EDOC 2009, pp. 108–117. IEEE Press, Piscataway (2009)
15. Ruehl, S.T., Andelfinger, U.: Applying software product lines to create customizable software-as-a-service applications. In: Proceedings of the 15th International Software Product Line Conference, SPLC 2011, vol. 2, pp. 16:1–16:4. ACM, New York (2011)
16. Scott, J., Kazman, R.: Realizing and refining architectural tactics: availability, Technical report, CMU/SEI-2009-TR-006 ESC-TR-2009-006 (2009)
17. Tekinerdogan, B., Ozturk, K., Dogru, A.: Modeling and reasoning about design alternatives of software as a service architectures. In: 2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 312–319, June 2011
18. Tool, P.: Puppet tool (retrieved, April 2016). http://puppetlabs.com/